Using Map-Reduce to Scale an Empirical Database

Shen Zhong (HT090423U)

Supervised by Professor *Y.C. Tay*

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2011

Using Map-Reduce to Scale an Empirical Database

Shen Zhong

shenzhong@comp.nus.edu.sg

Abstract

Datasets are crucial for testing in both industrial and academic fields. However, getting a dataset which has a proper size and can reflect the real data properties is not easy. Different from normal domain-specific benchmarks, UpSizeR is a tool that takes an empirical dataset $\mathcal{D}$ and a scale factor s as input and generates a synthetic dataset which keeps the properties of the original dataset but s times its size. UpSizeR is implemented using Map-Reduce which guarantees it could efficiently handle large datasets . In order to reduce I/O operations, we optimize our UpSizeR implementation to make it more efficient. We run queries on both the synthetic and the original datasets and compare the results to evaluate the similarity of both datasets.

ACKNOWLEDGEMENT

I would like to express my deep and sincere gratitude to my supervisor, Prof. Y.C. Tay. I am grateful for his invaluable support. His wide knowledge and his conscientious attitude of working set me a good example. His understanding and guidance have provided a good basis of my thesis. I would like to thank Wang Zhengkui. I really appreciate the help he gave me during the work. His enthusiasm in research has encouraged me a lot.

Finally, I would like to thank my parents for their endless love and support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SUMMARY

This thesis presents UpSizeR, a tool implemented using Map-Reduce, which takes an empirical relational dataset $\mathcal{D}$ and a scale factor s as input, and generates a synthetic dataset $\tilde{D}$ that is similar to $\mathcal{D}$ but s times its size. This tool can be used to scale up $\mathcal{D}$ for scalability testing ($s > 1$), scale down for application debugging ($s < 1$), and anonymization ($s = 1$).

UpSizeR's Algorithm describes how we extract properties (table size, degree distribution and dependency ratio etc.) from empirical dataset $\mathcal{D}$ and inject them into into synthetic dataset $\tilde{\mathcal{D}}$. We then give a Map-Reduce implementation which exactly follows each step described in the algorithm. This implementation is further optimized to reduce I/O operations and time consumption.

The similarity between $\mathcal{D}$ and $\tilde{\mathcal{D}}$ is measured using query results. To validate UpSizeR, we scale up a Flickr dataset and scale down a TPC-H benchmark dataset. The results show that the synthetic dataset is similar to the empirical dataset of the same size in terms of size of the query results. We also compare the time consumed by optimized and non-optimized UpSizeR. The results show the time consumption reduces by half using optimized UpSizeR. To test the scalability of UpSizeR, we

downsize a 200GB TPC-H dataset and upsize a 1GB dataset to 200GB. The results confirm that UpSizeR is able to handle both large input and large output datasets.

According to our study, we find most of the recent synthetic dataset generators are domain-specific, which cannot take advantage of the empirical dataset and may be misleading if we use those synthetic datasets as input of a specific DBMS. So we can hear the calling for application-specific benchmarks and see the early signs of them. We also study a parallel dataset generator and compare it with our UpSizeR.

Finally, we discuss the limitation of our UpSizeR tool and propose some directions in which we can improve our tool.

CHAPTER 1

INTRODUCTION

As a complex combination of hardware and software, a database management system (DBMS) needs sound and informative testing. The size of dataset and type of the queries affect the performance of the DBMS significantly. By this mean, we need a set of queries that may be frequently executed and a dataset of an appropriate size to test the performance of the DBMS, so that we can optimize the DBMS according to the results we get from the test. If we know what application the DBMS will be used for, we can easily get the set of queries. Getting the dataset of an appropriate size, however, is a big problem. One may have a dataset in hand, but it may be either too small or too large. Or one may have a dataset in hand which is not quite relevant to the application his product will be used for.

One possibility is to use a benchmark for the testing. A lot of benchmarks can provide domain-specific datasets which can be scaled to a desired size. As an example, consider the popular domain-specific TPC[3] benchmark: TPC-C is used for online transactions, TPC-H is designed for decision support, etc. Vendors could use these benchmarks to evaluate the effectiveness and robustness of their products, and researchers could use those products to analyze and compare their algorithms and prototypes. For these reasons, the TPC benchmarks have played

quite an important role in the growth of database industry and the progress of database research.

However, the synthetic data generated by the TPC benchmarks is often specialized. Since there is a tremendous variety of database applications, while there are only a few TPC benchmarks, one may not be able to find a TPC benchmark that is quite relevant to his application; furthermore, at any moment, there are numerous applications that are not covered by the benchmarks. In such cases, the results of the benchmarks can provide little information to indicate how well a particular system will handle a particular application. Such results are, at best, useless and, at worst, misleading.

Consider for instance, some new histogram techniques may be used for cardinality estimation (some recently proposed approaches include [9, 19, 29, 34]). Studying those techniques analytically is very difficult, because they often use heuristics to place buckets. Instead, it is a common practice to evaluate a new histogram by analyzing its efficiency and approximation quality with respect to a set of data distributions. By this means, the input datasets are very important for a meaningful validation. They must be carefully chosen to exhibit a wide range of patterns and characteristics. Multidimensional histograms are more complicated and require the validation datasets to be able to display varying degrees of column correlation and also different levels of skew in the number of duplicates per distinct value. Note that histograms are not only used to approximate the cardinality of range queries, but also to estimate the result size of complex queries that might have join and aggregation operations. Therefore, in order to have a a thorough validation of a new histogram technique, the designer needs to have a dataset whose data distributions have correlations that span over multiple tables(e.g., correlation between columns in different tables connected via foreign key joins). Such correlations are hard to

generated by purely synthetic methods, but can be found in empirical data.

Another example is analysis and measurement of online social networks, which have gained significant popularity recently. Using a domain-specific benchmark usually does not help since its data is usually generated independently and uniformly. The relation inside a table and among tables could never be reflected. For example, if the number of photos uploaded by a certain user is generated randomly, we cannot tell properties (such as heavy-tail) of the out degree from **User** table to **Photo** table. If the writer of comments and the uploader of photos are generated independently, we cannot reflect the correlations between the commenters of the photo and the uploder of the photo. In those cases, the structure of the social network could not be captured by such benchmarks, which means it is impossible to validate the power-law, small-world and scale-free properties using such synthetic data, let alone look into the structures of the social network. Although data could be crawled from internet and organized as tables, it is usually difficult to get a dataset with a proper size, while an in-depth analysis and understanding of a dataset big enough is necessary to evaluate current systems, and to understand the impact of online social networks on the Internet.

Automatic physical design for database systems (e.g., [12, 13, 35]) is also a problem that requires validation with carefully chosen datasets. Algorithms addressing this problem are rather complex and their recommendations crucially depend on the input databases. Therefore, it is suggested that the designer check whether the expected behavior of a new approach (both in terms of scalability and quality of recommendations) is met for a great range of scenarios. For that purpose, test cases should not be simplistic, but instead exhibit complex intra- and inter-table correlations. As an example, consider the popular TPC-H benchmark. Although the schema of TPC-H is rich and the syntactical workloads are complex, the resulting

data is mostly uniform and independent. We may ask how would recommendations change if the data distribution shows different characteristics in the context of physical database design. What if the number of **orders** per **customer** follows a Possion distribution? What if **customers** buy **lineitems** that are supplied only by vendors in their own **nation**? What if **customer** balances depend on the total price of their respective open **orders**? Dependencies across table must be captured to keep those constraints.

UpSizeR is a tool that aims to capture and replicate the data distribution and dependencies across tables. According to the properties captured from the original database, it generates a new database with demanded size and with inter- and intra-table correlations kept. In other words, it generates a database similar to the original database with a specified size.

Generating Dataset Using Map-Reduce

UpSizeR is a scaling tool presented by Tay et al.[33] for running on a single database server. However, the dataset size it can handle is limited by the memory size. For example, it is impossible for computers with 4 GB memory to scale down a 40 GB dataset using the memory based UpSizeR. Instead, we aim to provide a non-memory based and efficient UpSizeR tool that can be easily deployed on any affordable PC-based cluster.

With the dramatic growth of internet data, terabyte size databases become fairly common. It is necessary for a synthetic database generator to be able to cope with such large datasets. Since we are generating synthetic databases according to empirical databases, our tool needs to handle both large input and large output. Memory based algorithms are not able to analyze large input datasets. Normal disk based algorithms are too time-consuming. So we need a non-memory based

parallel algorithm to implement UpSizeR.

A promising solution is to use cloud computing, which is adopted by us. There are already low-cost commercially available cloud platforms (e.g., Amazon Elastic Compute Cloud (Amazon EC2)) where our techniques can be easily deployed and made accessible to all. End-users may also be attracted by the pay-as-you-use model of such commercial platforms.

Map-Reduce has been widely used in many different applications. This is because it is highly scalable and load balanced. In our case, when analyzing an input dataset, Map-Reduce can split the input and assign each small piece to the processing unit, and then finally results are automatically merged together. When generating a new dataset, each processing unit reads from a shared file system and generates its own part of tuples. This makes UpSizeR a scalable and time-saving tool.

Using Map-Reduce to implement UpSizeR involves two major challenges:

1. How can we develop an algorithm suitable for Map-Reduce implementation?

2. How can we optimize the algorithm to make it more efficient?

Consider the first challenge: There are a lot of limitations for doing computation on the Map-Reduce platform. For example, it is difficult to generate unique values (such as primary key values) because each Map-Reduce node cannot communicate with each other when it is working. Besides, quite different from memory based algorithm which organize data as structures or objects in memory, Map-Reduce must organize data as tuples in files. Each Map-Reduce node reads in a chunk of data from file and processes one tuple at a time, making it difficult to randomly pick out a tuple according to a field value in the tuple. Moreover, we must consider how to break down UpSizeR into small Map-Reduce tasks and how to manage

the intermediate results between each task. The solutions of these problems are described in Sec. 4.3.

Consider the second challenge: Although Map-Reduce nodes can process in parallel, reading from and writing into disks still consumes a lot of time. In order to save time, we must reduce I/O operations and reduce intermediate results. We manage this by merging small Map-Reduce tasks into one task, doing as much as we can in a Map-Reduce task. We describe the optimization in Sec. 4.4.

Migrating into Map-Reduce platform should keep the functionality of UpSizeR. We tested UpSizeR using Flickr and TPC-H datasets. The results confirm that the synthetic dataset generated by our tool is similar to the original empirical dataset in terns of query result size.

CHAPTER 2

PRELIMINARY

In this chapter, we introduce the preliminaries of our UpSizeR tool. In Sec. 2.1 we state the problem UpSizeR deals with and the motivation of UpSizeR. In Sec. 2.2 and 2.3 we introduce our implementation tool MapReduce.

2.1 Introduction to UpSizeR

2.1.1 Problem Statement

We aim to provide a tool to help database owners generate application-specific datasets of specific size. We state this issue as the **Dataset Scaling Problem**:

Given a set of relational tables $\mathcal{D}$ and a scale factor s, generate a database state $\tilde{\mathcal{D}}$ that is similar to $\mathcal{D}$ but s times its size.

This thesis presents **UpSizeR**, a first-cut tool for solving the above problem using cloud computing.

Here we define scale factor s in terms of number of tuples. However, it is not necessary to stick to numerical precision. For example, suppose $s = 10$, it is acceptable if we generate a synthetic dataset $\tilde{\mathcal{D}}$ that is 10.1 times $\mathcal{D}$'s size. Usually, if the table has no foreign key, we will generate number of tuples exactly s times the

original corresponding table. The other tables will be generated based on tables that are already generated and according to the properties we extracted, so that it would be around s times the original corresponding tables.

Rather, the most important definition here is "similarity". The definition of "similarity" can be used in two scenarios: (1)How can we generate $\tilde{\mathcal{D}}$ that is similar to $\mathcal{D}$? We manage this by extracting properties from $\mathcal{D}$ and injecting them into $\tilde{\mathcal{D}}$. (2)How can we validate the similarity between $\tilde{\mathcal{D}}$ and $\mathcal{D}$? We say $\tilde{\mathcal{D}}$ is similar to $\mathcal{D}$ if $\tilde{\mathcal{D}}$ can reflect relationships among the columns and rows of $\mathcal{D}$. We don't measure similarity by the data itself (e.g. doing statistical test or extracting graph properties), because we use these properties to generate $\tilde{\mathcal{D}}$. Instead, we use results of queries (in this thesis we use query result size and aggregated values) to evaluate the similarity, because those information is enough to understand the properties of the datasets and to analyze the performance of a given DBMS.

2.1.2 Motivation

We could scale an empirical dataset in three directions: scale up ($s > 1$), scale down ($s < 1$) and equally scale ($s = 1$). The reason why one might want to synthetically scale an empirical dataset also varies with different scale factors:

There are various purposes for scaling up a dataset. The user populations of some web applications are growing at breakneck speed (e.g. Animoto[1]), as we can see that even datasets of terabyes could be small in nowadays. However, one may not have a dataset big enough, so a small but fast growing service may need to test the scalability of their hardware and software architecture with larger versions of their datasets. Another example is where a vendor only gets a sample of the dataset he bought from an enterprise (e.g. it is not convenient to get the entire dataset). The vendor can scale up the sample to get the dataset of desired

size. Consider a more common case that we usually crawl data from Internet for analysing social network and testing the performance of certain DBMS. This is a quite time consuming operation. However, if we have a dataset big enough to capture the statistical property of the data, then we can use UpSizeR to scale the dataset into desired size.

Scenarios that we need to down scale a dataset also commonly exist. One may want to take a small sample of a large dataset. But this is not a trivial operation. Consider this example: if we have a dataset with 1000000 employees, and we need a sample having only 1000 employees. Randomly picking 1000 employees is not sufficient. Since employee may refer to or be referred by other tables and we need to recursively pick tuples in other tables accordingly. The resulting dataset size is out of control because of this recursively adding. Besides, because the sample we get may not capture the properties of the whole dataset, the resulting dataset may not be able to reflect the original dataset. Instead, the problem can be solved by downsizing the dataset using UpSizeR with $s < 1$. Even for an enterprise itself may want to downsize its dataset. For example, running a production dataset for debugging a new application may be too time consuming, one may want to get a small synthetic copy of its original dataset for testing.

One may feel surprised why we need to scale a dataset with $s = 1$. However, if we take privacy or proprietary information into consideration, such scaling will make sense. As the users don't want to leak their privacy, the use of production data which contains sensitive information in application testing requires that the production data be anonymized first. The task of anonymizing production data becomes difficult since it usually consists of constraints which must also be satisfied in the anonymized data. UpSizeR can also address such issues, since the output dataset is synthetic. Thus, UpSizeR can be viewed as an anonymization tool for

$s = 1$.

2.2 Introduction to Map-Reduce

Map-Reduce is a programming model and associated implementation for processing and generating large dataset. The fundamental goal of Map-Reduce is providing a simple and powerful interface for programmers to automatically distribute and parallelize a large scale computation. It is originally designed for large clusters of commodity PCs, but it can also be applied on Chip Multi-Processor (CMP) or Symmetric Multi-Processing (SMP) computers.

The idea of Map-Reduce comes from the observation that the computation of certain datasets always take a set of input key/value pairs and produces a set of output key/values pairs. So the computation is always based on some key, e.g. compute the occurrence of some key words, etc. So the map function will gather the pairs that have the same key value together and store them into some place, the reduce function reads in those intermediate pairs, which have all the values of some keys, does the computation and writes down the final results. For example, suppose we want to count the appearance of each different word in a set of documents. We will use these documents as input, the map function will pick out each single word and emit intermediate tuple with the word as key. Tuples with the same key value will be gathered to the reducers. The reduce function will count the occurrence of each word and emit the result using the word as key and the number of tuples having this key as value.

Performance can be improved by partitioning the task into subtasks of different size, if the computing environment is heterogeneous. Suppose the nodes in the computing environment have different processing ability, we can give more tasks to

more powerful nodes, so that all nodes can finish their tasks in roughly the same time. In this case, the computing elements are made better use of, eliminating the bottleneck.

2.3 Map-Reduce Architecture and Computational Paradigm

Map − Reduce architecture : There are two kinds of nodes under the Map-Reduce framework: Namenode and Datanode. The NameNode is a master of the file system. It takes charge of spliting data into blocks and distributing the blocks to the data nodes (DataNodes) with replication for fault tolerance. A JobTracker running on the NameNode keeps track of the job information, job execution and fault tolerance of jobs executing in the cluster. The NameNode can split the submitted job into multiple tasks and assign each task to a DataNode to process.

The DataNode stores and processes the data blocks assigned by the NameNode. A TaskTracker running on the DataNode communicates with the JobTracker and tracks the task execution.

Map − Reduce computational paradigm : The Map-Reduce computational paradigm can parallelize the job processing by dividing it into small tasks, each of which is assigned to a different node. The computation of Map-Reduce follows a fixed model with a map phase followed by the reduce phase. The data is split by the Map-Reduce library into chunks, which is further distributed to the processing units (called mapper) on different nodes. The mapper reads the data from the file system, processes it locally, and then emits a set of intermediate results. The intermediate results are shuffled according to the keys, and delivered to the next processing unit (called reducer). Users can set their own computation logic

by writing the map and reduce functions in their applications.

Map phase : Each DataNode has a map function which processes the data chunk assigned to it. The map function reads in the data as the form of $(key, value)$ pairs, does computation on those $(k1, v1)$ pairs and transforms them into a set of intermediate $(k2, v2)$ pairs. The Map-Reduce library will sort and partition all the intermediate pairs and pass them to the reducers.

Shuffling phase : The Map-Reduce library has a partition function which gathers the intermediate $(k2, v2)$ pairs emitted by the map function and partitions them into M pieces stored in the file system, where M is the number of reducers. Those pieces of pairs are then shuffled and assigned to the corresponding reducers. Users can specify their own partitioning function or use the default one.

Reduce phase : The reducer receives a sorted value list consisting of intermediate pairs $(k2, v2)$ with the same key that are shuffled from different mappers. It makes a further computation to the key and values and produces new $(k3, v3)$ pairs which are the final results written to the file system.

CHAPTER 3

SPECIFICATION

In this chapter , we first fix our terminology and notation in Sec. 3.1, list and analyze our assumptions in Sec. 3.2. Input and output for UpSizeR are described in Sec. 3.3.

3.1 Terminology and Notation

We assume the readers are already familiar with some basic terminologies, such as database, primary key, foreign key, etc. We introduce our choice of terminology and notation as following.

In the **relational data model**, a database state $\mathcal{D}$ records and expresses a **relation** which consists of a **relation schema** and a **relation instance**. The relation instance is a **table**, and the relation schema describes the **attribute**s, including a **primary key**, for the table. A table is a set of **tuple**s, in which each tuple has the same attributes as the relation schema. We call table T as **static** table if T's content should not change after scaling.

We call an attribute K a **foreign key** of table T if it refers to a primary key K' of table T' The foreign key relationship defines an **edge** between T and T', pointing from K to K'. The tables and the edges form a directed **schema graph**

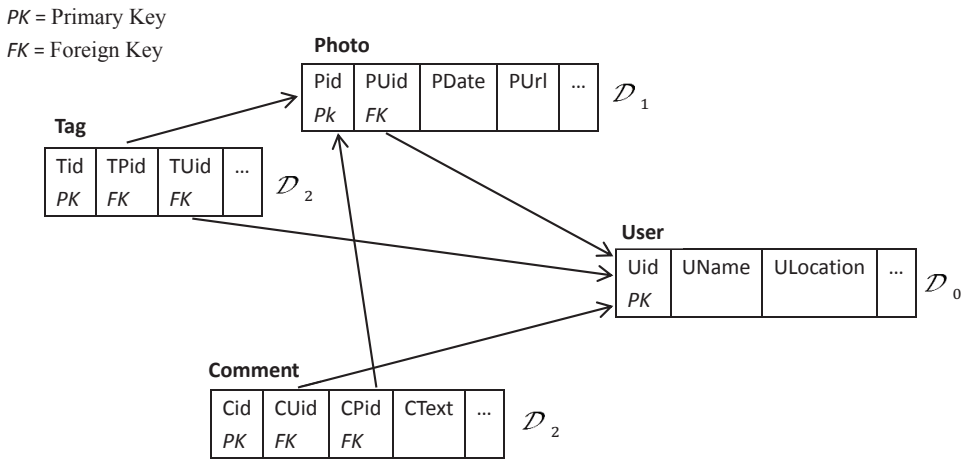

Figure 3.1: A small schema graph for a photograph database $\mathcal{F}$.

for $\mathcal{D}$.

Fig. 3.1 gives an example of a schema graph for a database $\mathcal{F}$, like Flickr, that stores photographs uploaded by, commented upon and tagged by a community of users.

Each edge in the schema graph induces a bipartite graph between T and T', with bipartite edges between a tuple in T with K value v and the tuples in T' with K' value v. The number of edges from T to T' is the out degree of value v in T, we use $deg(v, T')$ to denote such degree. This is illustrated in Fig. 3.2 for $\mathcal{F}$.

A **scale factor** s needs to be provided beforehand. To scale $\mathcal{D}$ is to generate a synthetic database state $\tilde{\mathcal{D}}$ such that:

S1 $\tilde{\mathcal{D}}$ has the same schema as $\mathcal{D}$.

S2 $\tilde{\mathcal{D}}$ is similar to $\mathcal{D}$ in terms of query results.

S3 For each non-static table T_0 that has no foreign key, the number of T_0 tuples in $\tilde{\mathcal{D}}$ should be s times that in $\mathcal{D}$; the sizes of non-static tables with foreign keys are indirectly determined through their foreign key constrains.

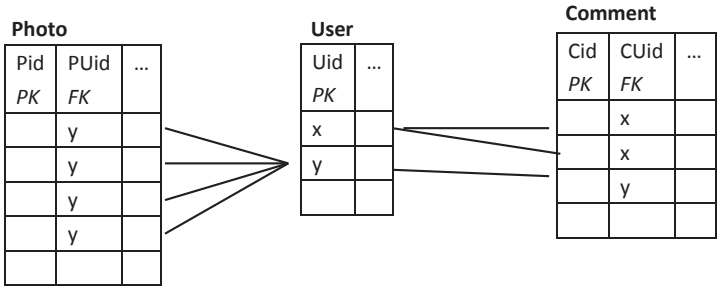

Figure 3.2: A schema graph edge in Fig. 3.1 from **Photo** to **User** for the key **Uid** induces a bipartite graph between the tuples of **User** and **Photo**. Here $deg(x, \textbf{Photo}) = 0$ and $deg(y, \textbf{Photo}) = 4$, Similarly, $deg(x, \textbf{Comment}) = 2$ and $deg(y, \textbf{Comment}) = 1$

S4 The content of static table does not change after scaling.

The most important definition should be **similarity**. How should we measure the similarity between $\tilde{\mathcal{D}}$ and $\mathcal{D}$? We choose not to measure the similarity by data itself (e.g. statistical test or graph property). This is because we extract such properties from the original dataset and apply them into the synthetic dataset, which means those properties will be kept in the synthetic dataset. Rather, since our motivation for UpSizeR lies in its use for scalability studies, UpSizeR should provide accurate forecasts of storage requirement, query time and retrieval results for larger datasets. So we could use the latter two as the measurement of similarity, and they require some set $\mathcal{Q}$ of test queries.

Therefore, in addition to the original database state $\mathcal{D}$, such a set of queries is supposed to be owned by the UpSizeR users. By running the queries, the user records the tuples retrieved and the aggregates computed to measure the similarity between $\mathcal{D}$ and $\tilde{\mathcal{D}}$. Since the queries are user specified and are designed for testing a certain application, our definition of similarity makes (S2) application-specific.

We explain (S3) using the schema shown in Fig. 3.1. Table **User** does not

have foreign keys. Suppose in the original dataset $\mathcal{D}$, the number of tuples of **User** is n, we will generate $s * n$ tuples for **User** in $\tilde{\mathcal{D}}$. We generate table **Photo** in $\tilde{\mathcal{D}}$ according to the generated **User** table and $deg(\mathbf{Uid}, \mathbf{Photo})$. **Comment** has two foreign keys: **CPid** and **CUid**. So its size is determined by the synthetic **Photo** and **User** table, and the correlated values of $deg(\mathbf{Uid}, \mathbf{Comment})$ and $deg(\mathbf{Pid}, \mathbf{Comment})$.

In order to scale a database state $\mathcal{D}$, we need to extract data distribution and dependency property of $\mathcal{D}$. To capture those properties, we need to introduce the following notations.

Degree Distribution

This statistical distribution is used to capture inter-table correlations and data distribution of the empirical database. Suppose K is a primary key of table T_0, let $T_1, \ldots T_r$ be the tables who reference K as their foreign key. We use $deg(v, T_i)$ to denote the out degree of a K value v to table T_i, as is described in Fig. 3.2. We use $Fr(deg(K, T_i) = d_i)$ to denote the number of K values whose out degree from T_0 to T_i is d_i. The we can define the joint degree distribution f_K as:

$$f_K(d_1, \ldots, d_r) = Fr(deg(K, T_1) = d_1, \ldots, deg(K, T_r) = d_r)$$

For example, there are 100 users uploaded 20 photos in the empirical database. Among those users, 50 wrote 200 comments. Then we can record

$$Fr(deg(\mathbf{Uid}, \mathbf{Photo}) = 20, deg(\mathbf{Uid}, \mathbf{Comment}) = 200) = 50;$$

By keeping joint degree distribution we can keep not only the data distribution, but the relation of tables that are established by having the same foreign key.

For example, it is a common phenomenon that the more photo one uploads, the more comments he is likely to write. This property is kept because the conditional probability $Pr(deg(\mathbf{Uid}, \mathbf{Photo})|deg(\mathbf{Uid}, \mathbf{Comment}))$ is kept.

Dependency Ratio

Looking at the schema graph in Fig. 3.1, we may find such a triangle: **User**, **Photo** and **Comment**. Both table **Photo** and **Comment** refer to primary key **Uid** of table **User** as their foreign key. Meanwhile, table **Comment** refers to primary key **Pid** of table **Photo** as its foreign key. We say table **Comment depends on** table **Photo**, because **Comment** refers to **Photo**'s primary key as its foreign key and **Photo** is generated before **Comment**. From each tuple in table **Photo** we can find such $<\mathbf{Pid}, \mathbf{Uid}>$ pair, of which **Pid** is the primary key of **Photo** and **Uid** is the foreign key of **Photo**. In table **Comment** we can also find such pairs, both elements of which are foreign keys. If we can find a tuple in **Comment**, the pair value of which could be found in the tuples of **Photo**, we say this tuple in **Comment** depends on the corresponding tuple in **Photo** and this **Comment** tuple is called a **dependent tuple**.

In the empirical database, we calculate the number of dependent tuples as **dependency number**. We define **dependency ratio** as **dependency number/table size**. As can be seen in Sec. 3.2, we assume the dependency ratio does not change with the size of the dataset. In the synthetic database, we generate s times the original dependent tuples.

This metric capture both inter- and intra-table relationship. For example, a lot of users like to comment on their own photos. If a user comments on his own photo, we may find such a dependent tuple in **Comment** whose **Uid** and **Pid** value appears in **Photo** as primary key and foreign key respectively. By keeping

Photo

Pid *PK*	PUid *FK*	...
a	x	
b	x	
c	y	
d	y	
e	z	

Comment

Cid *PK*	CUid *FK*	CPid *FK*	...
1	x	a	
2	x	c	
3	y	d	
4	z	e	
5	x	a	
6	x	a	
7	x	c	

Figure 3.3: A table content graph of **Photo** and **Comment**, in which **Comment** depends on **Photo**

.

dependency ratio, we can keep this property of the original database. In Fig. 3.3, Tuple $< 1, x, a >$, $< 3, y, d >$, $< 4, z, e >$, $< 5, x, a >$ and $< 6, x, a >$ in **Comment** are dependent tuples. They depend on tuple $< a, x >$, $< d, y >$ and $< e, z >$ in **Photo**, and we say the dependency number of **Comment** is 5 and dependency ratio is 5/7.

Finally, we refer to generation of values for non-key attributes as **content generation**

We will use v, T and $deg(v, T')$ to denote a value, table and degree in given $\mathcal{D}$, and $\tilde{v}$, $\tilde{T}$, and $deg(\tilde{v}, \tilde{T})$ to denote their synthetically generated counterparts in $\tilde{\mathcal{D}}$.

3.2 Assumptions

We made the following assumptions in our implementation of UpSizeR.

A1. Each primary key is a singleton attribute.

A2. The schema graph is acyclic.

A3. Non-key attribute values for a tuple t only depends on the key values.

A4. Key values only depend on the joint degree distribution and dependency ratio.

A5. Properties extracted do not change with the dataset size.

In our UpSizeR's implementation, we have the above 5 assumptions. (A3) says we only care about the relationship among key values. (A4) means the properties we extracted from the original dataset are degree distribution and dependency ratio. (A5) talks about both degree distribution and dependency ratio. For degree distribution, we assume it is static. Take Flickr dataset as an example, we assume the number of comments per user has the same distribution in $\mathcal{F}$ and $\tilde{\mathcal{F}}$. We also assume the dependency ratio does not change with the size of the dataset, which means the dependency number of a table in a synthetic dataset becomes s times the dependency number of the original table. In our Flickr example, we assume the number of users who comments on his/her own graph increases with the number of users.

3.3 Input and Output

The input to UpSizeR is given by an empirical dataset $\mathcal{D}$ and a positive number s which specifies the **scale factor**.

In response, a syntactic database state $\tilde{\mathcal{D}}$ will be generated by UpSizeR as output, satisfying (S1), (S2) and (S3) - see Sec. 3.1. The size of $\tilde{\mathcal{D}}$ is only approximately s times the size of $\mathcal{D}$. This is because some tables may be static, the size of which may not change; the size of some table may be determined by key constraints; and there are some randomness in tuple generation.

In the Dataset Scaling Problem, the most important issue is similarity. Since we aim to provide an application-specific dataset generator, we must provide an application-specific standard to define the similarity for UpSizeR to be general

applicable. Using query results (instead of, say, graph properties or statistical distribution) to measure the similarity, as is described in (S2), provides a solution to the UpSizeR user.

CHAPTER 4

PARALLEL UPSIZER ALGORITHMS AND DATA FLOW

In this chapter, we introduce the algorithms and implementation of UpSizeR. In Sec. 4.1 we introduce properties extracted from original dataset and how we apply them into synthetic dataset. In Sec. 4.2 we describe the basic algorithms of UpSizeR. In Sec. 4.3 we describe how we implement UpSizeR and make it suitable for Map-Reduce platform. In Sec. 4.4 we describe how we optimize UpSizeR to reduce I/O operations and time consumption.

4.1 Property Extracted from Original Dataset

We first extract properties from the original dataset, and then apply those properties into the synthetic dataset. What properties to extract significantly affects the similarity between the empirical database and the synthetic database. Here we introduce the properties we extract and how those properties are kept as follows:

Table Size

Table size is the number of tuples in each table. As is described in (S3), for a non-static tables without foreign keys, the number of tuples we generate should be s times that of the original table, and in (S4) we say a table is static if its content does not change after scaling. Suppose the number of tuples in table T is n, we keep this property by generating $s * n$ unique primary keys in $\tilde{T}$ if T is not static. If T is static, we will generate n tuples in $\tilde{T}$.

Joint Degree Distribution

Suppose T is a table whose primary key K is referenced by $T_1, \ldots, T_r$. We calculate such tuple:

$$< deg(K, T_1), \ldots, deg(K, T_r), Fr >$$

In which, $deg(v, T_i)$ is the out degree from T to T_i, $(1 \leq i \leq r)$. Fr is the number of primary key values (frequency) that have such degrees. According to (A3), the degree distribution is static, we do not change each degree value unless T is static. Note that the degree distribution is static means that the out degree of each primary key value in T remains the same in $\tilde{T}$, while a table T is static indicates that the content of T remains the same in $\tilde{T}$.

We use such degree frequency tuples to generate the degrees of each primary key value in $\tilde{T}$ when generating new tables. If neither T nor T_i is static, Fr is multiplied by s and $deg(K, T_i)$ remains the same. If T is static and T_i is non-static, Fr remains the same and $deg(K, T_i)$ is multiplied by s. If both T and T_i are static, both Fr and $deg(K, T_i)$ remain the same. For example, suppose we have such a degree frequency tuple $< deg(K, T_1) = 50, Fr = 10 >$ and $s = 2$. If neither T nor T_i is static, we will choose 20 tuples in $\tilde{T}$ and set the degree of the primary key

values in those tuples to be 50. If T is static and T_i is non-static, we will choose 10 tuples in $\tilde{T}$ and set the degree of the primary key values in those tuples to be 100. If both T and T_i are static, we will choose 10 tuples in $\tilde{T}$ and set the degree of the primary key values in those tuples to be 50.

Dependency Ratio

We compute dependency ratio for each table that depends on the table. Since dependency ratio does not change, the number of dependent tuples increases with the increase of table size. Suppose one table T depends on another table, the number of dependent tuples is n, we will generate $s * n$ dependent tuples when we generate $\tilde{T}$.

4.2 UpSizeR Algorithms

In this section, we describe the basic UpSizeR algorithms together with pseudo-code, using $\mathcal{F}$ as an example.

4.2.1 UpSizeR's Main Algorithm

First, we need to sort the table and group them into subsets. This is because some tables refer to other tables' primary key as foreign keys. We must generate those being referenced first. After that we extract degree distribution and dependency ratio from the original dataset. Using those information, we generate the tables in each subset.

Algorithm 1: UpSizeR main algorithm

Data: database state $\mathcal{D}$ and a scale factor s
Result: a synthetic database state that scales up $\mathcal{D}$ by s

use schema graph to sort $\mathcal{D}$ into $\mathcal{D}_0$, $\mathcal{D}_1$, $\mathcal{D}_2$, ...;
get joint degree distribution f_K from $\mathcal{D}$ for each key K;
get dependency ratio for each table that depends on other table;
foreach $T \in \mathcal{D}_0$ **do**
    generate $\tilde{T}$;
$i = 0$;
repeat
    $i = i + 1$;
    foreach $T \in \mathcal{D}_i$ **do**
        flag(T) = false;
    forall the $T \in \mathcal{D}_i$ *and flag(T) = false* **do**
        generate table $\tilde{T}$;
        flag(T) = true;
until *all tables are generated*;

4.2.2 Sort the Tables

Recall from (A2), we assume that the schema graph is acyclic. UpSizeR firstly groups the tables in $\mathcal{D}$ into subsets $\mathcal{D}_0$, $\mathcal{D}_1$, $\mathcal{D}_2$, ... by sorting this graph, in the following sense:

- all tables in $\mathcal{D}_0$ have no foreign key.
- for $i \geq 1$, $\mathcal{D}_i$ contains tables whose foreign keys are primary keys in $\mathcal{D}_0 \cup \mathcal{D}_1 \cup \ldots \cup \mathcal{D}_{i-1}$

For $\mathcal{F}$, $\mathcal{D}_0 = \{\textbf{User}\}$, $\mathcal{D}_1 = \{\textbf{Photo}\}$ and $\mathcal{D}_2 = \{\textbf{Comment, Tag}\}$; here tables in $\mathcal{D}_i$ coincidentally have i foreign keys. This is not true in general.

4.2.3 Extract Probability Distribution

For each table T that is referenced by other tables in $\mathcal{D}$, UpSizeR processes T to extract the joint degree distribution f_K, where K is the primary key of T (see Sec.

Algorithm 2: Sort the tables

Data: database state $\mathcal{D}$
Result: sorted database states $\mathcal{D}_0$, $\mathcal{D}_1$, $\mathcal{D}_2$, ...

```
1  i = 0;
2  while D is not empty do
3      foreach table T in D do
4          if T does not have a foreign key then
5              add T into D_0;
6              remove T from D;
7          else if every foreign key in T is a primary key of tables in D_0, ..., D_i then
8              add T into D_{i+1};
9              remove T from D;
10     i = i + 1;
```

3.1). We use f_K to generate new foreign key degree $deg(\tilde{v}, \tilde{T}_i)$, where $\tilde{T}_i$ is any table with K as its foreign key, when generating new database state $\tilde{\mathcal{D}}$. The conditional degree distribution is kept since we use the joint degree distribution,

The algorithm is quite simple, which can be seen from Sec. 3.1. The details of generating the joint degree distribution using Map-Reduce will be described in Sec. 4.3.

4.2.4 Generate Degree

After getting the degree distribution, we need to exact degree for each primary key that is referenced by other tables. In our $\mathcal{F}$ example, deg(**Uid, photo**) and deg(**Uid, Comment**) are correlated, since they refer to the same table as foreign key. We must catch the conditional probability:

$$Pr(deg(\textbf{Uid, Comment}) = d' \mid deg(\textbf{Uid,Photo}) = d)$$

so that we can explain the phenomenon that users who upload more photos are likely to write more comments.

Since we have already got the joint degree distribution, it is easy to keep such conditional probability. For example, if T's primary key K is referenced by T_1 and T_2, and we have such degree distribution tuple: $< deg(K, T_1),\ deg(K, T_2),\ Fr >$. We will generate Fr primary key values whose degree of T_1 and T_2 is assigned to be $deg(K, T_1)$ and $deg(K, T_2)$ respectively.

4.2.5 Calculate and Apply Dependency Ratio

Recall from Sec. 3.1, we say T depends on T' if T has two foreign keys FK_1 and FK_2, in which FK_1 refers to T''s primary key and FK_2 refers to the same table as T''s foreign key does. In order to calculate dependency ratio, we only need to figure out the dependency number, which is the number of tuples in T having $< FK_1, FK_2 >$ pairs that appear in T' as primary key and foreign key values, after knowing the table size. The detail algorithm of calculating dependency number using Map-Reduce will be shown in Sec. 4.3.

We want to keep the dependency ratio in our synthetic database. This means: if the number of dependent tuples in T is d, we need to generate $d * s$ dependent tuples in $\tilde{T}$. We also need to make sure that the degree of each foreign key in $\tilde{T}$ matches the degree distribution. So we use the degree we generated for each foreign key in $\tilde{T}$, generated table $\tilde{T}'$ and number dependent tuples d in $\tilde{T}$ as input, generating such dependency tuple:

$$< pair\ < FK_1, FK2 >,\ pair\ degree,\ left\ degree\ FK_1,\ left\ degree\ FK_2 >$$

In which, $pair\ < FK_1, FK_2 >$ appears in $\tilde{T}'$, $pair\ degree$ is $min\{deg(FK_1, \tilde{T}), deg(FK_2, \tilde{T})\}$,

left degree FK_1 is $deg(FK_1, \tilde{T}) - $ *pair degree*, *left degree* FK_2 is $deg(FK_2, \tilde{T}) -$ *pair degree*.

Algorithm 3: Generate dependency tuples

Data: generated table $\tilde{T}'$, generated degree, number dependent tuples d
Result: dependency tuples

```
1 i = 0;
2 foreach foreign key value pair < v1, v2 > which appears in T̃' do
3     if i < d then
4         generate
          pair < v1, v2 >, pair degree, left degree v1, left degree v2;
5         i+ = pair degree;
6     else
7         generate pair < v1, v2 >, 0, deg(v1, T̃), deg(v2, T̃);
8     if deg(v2, T̃) > 0 but v2 does not appear in T̃' as a foreign key then
9         generate pair < 0, v2 >, 0, 0, , deg(v2, T̃);
```

After getting such dependency tuples, when we generate table $\tilde{T}$, we will generate tuples with such value pair according to the *pair degree*, the other foreign key values are randomly combined with each other according to *left degree*. The details are described in Sec. 4.2.8.

4.2.6 Generate Tables without Foreign Keys

Suppose T in $\mathcal{D}_0$ has h tuples. Since T has no foreign keys, UpSizeR simply generate $s * h$ primary key values for $\tilde{T}$. For example, the **User** has s times the number of tuples of **Uid** in $\mathcal{F}$.

Recall assumption (A4), that non-key values of a tuple depend only on its key values. For $\mathcal{D}_0$ this means that the non-key value attributes can be independently generated (without regard to the primary key values, which are arbitrary) by some content generator.

For example, values for **UName** and **ULocation** in $\tilde{\mathcal{F}}$ can be picked from sets of names and locations, according to frequency distributions extracted from $\mathcal{F}$.

4.2.7 Generate Tables with One Foreign Key

Suppose T' has foreign key set $\mathbf{K} = \{K\}$, where K is primary key of table T. In the $\mathcal{F}$ example, **Photo** has $\mathbf{K} = \{\mathbf{Uid}\}$ and **User** is generated first; for each **Uid** value $\tilde{v}$, we then generate $deg(\tilde{v}, \mathbf{Photo})$ tuples for **Photo** using $\tilde{v}$ as its foreign key value.

In general, for each $\tilde{v}$, we generate $deg(\tilde{v}, \tilde{T'})$ tuples of $\tilde{T'}$, using $\tilde{v}$ as their K value and arbitrary (but unique) values for their primary key. Each tuple's non-key value are then assigned by content generation.

Algorithm 4: Generate table with one foreign key

Data: degree generated for primary key K
Result: a synthetic table with K as its primary key

```
1 foreach K value ṽ do
2     generate degree(ṽ, T̃′) tuples with ṽ as their foreign key value, keeping
      the primary key value unique;
3     generate non-key contents;
4     form a tuple using the primary key and non-key values;
```

4.2.8 Generate Dependent Tables with Two Foreign Keys

Suppose T' has foreign key set $\mathbf{K} = \{K_1, K_2\}$ and depends on table T. For $\mathcal{F}$, **Comment** has $\mathbf{K} = \{\mathbf{Pid}, \mathbf{Uid}\}$ and depends on **Photo**.

We generate such tables in the following two steps:

Generate dependent tuples: In generating dependency ratio part, we get foreign key pairs and the degree of such pairs. We will generate *pair degree* tuples with the corresponding pair value as foreign keys. The implementation is similar

to generating tables with one foreign key.

For example, we have a dependency ratio tuple for **Comment**, in which the *pair value* is $< 100, 80 >$ and the *pair degree* is 50, we will generate 50 tuples in the synthetic **Comment** table with 100 and 80 as their **CPid** and **CUid** value respectively.

Generate non-dependent tuples: We generate non-dependent tuples according to $left\ degree$ of each foreign key K_i. First, we generate foreign key values for each K_i separately according to $left\ degree$. Then we randomly combine those foreign keys and add a unique primary key value to form a tuple. The non-key value are generated by content generator.

For example, after generating the dependent tuples, we still have a **CUid** value 100 whose $left\ degree$ is 10, and two **CPid** value 20 and 30 whose $left\ degree$ are 3 and 7 respectively. We will generate 3 tuples in the synthetic **Comment** table using 20 as their **CPid**, 100 as their **CUid** and 7 tuples using 30 as their **CPid** and 100 as their **CUid**.

Algorithm 5: Generate dependent table with two foreign keys

Data: dependency ratio tuples for table T'
Result: tuples for table T'

```
foreach pair value < v1, v2 > do
    generate degree(pair < v1, v2 >) tuples with the pair value as their
    foreign key values, keeping the primary key value unique;
    generate content of non-key;
    form a tuple;
foreach K1 value whose left degree is larger than 0 do
    randomly choose K2 value that is not used according to left degree;
    choose a unique primary key;
    generate content of non-key values;
    form a tuple;
```

4.2.9 Generate Non-dependent Tables with More than One Foreign Key

The algorithm of generating non-dependent tables with more than one foreign keys is similar with generating non-dependent tuples of dependent tables. We randomly choose foreign key values according to degree generated before, and assign a primary key for the tuple. Then we generate non-key values using content generator. In $\mathcal{F}$, we don't have such example, but it is common in real practice. For example, in the TPC-H benchmark, table **PARTSUPP** has two foreign keys: **PS_PARTKEY** and **PS_SUPPKEY**. But it does not depend on any table.

4.3 Map-Reduce Implementation

We use Map-Reduce to do the computations and statistics on large dataset. In UpSizeR, we use Map-Reduce in those parts: compute table size, build degree distribution, generate degree, compute dependency ratio, generate tables without foreign key, generate tables with one foreign key, generate dependent tables with two foreign keys, and generate non-dependent tables with more than one foreign key. In the following sub-sections, we introduce the parallel algorithms and data flow.

4.3.1 Compute Table Size

Before generating the tables, we need to compute the table size (number of tuples in each table) to figure out how many tuples we need to generate in each table. The user can provide the table size if he knows before hand. But if it is not provided, we need to compute it using Map-Reduce. In this step, we read through each table file and record how many tuples are there in the file. It is a simple *Word Count*

task, so we omit the data flow and the pseudo code.

4.3.2 Build Degree Distribution

Given a primary key K of table T that is referenced by $T_1, T_2, \ldots$, we compute the degree distribution of T in 3 steps:

1. For each primary key value v, we compute the number of its occurrence in T_i.
   In this step we get such result tuple: $< primary\ key\ value\ v,\ name\ of\ T_i,\ deg(v, T_i) >$.
   We call this step $Value\ Count$.

2. For each primary key value v, we collect the degree of v in each table T_i. In this step, we get such result tuple: $< primary\ key\ value\ v,\ deg(v, T_1),\ deg(v, T_2), \ldots >$
   We call this step $Value\ Gather$.

3. We get degree distribution: $< deg(v, T_1),\ deg(v, T_2), \ldots,\ Fr >$ using results got from last step as input. We call this step *Build Degree Distribution*.

The data flow is shown in Fig. 4.1, and pseudo code is shown in Fig. 4.2. For example, **User** is referred by **Comment** and **Photo**. Then $Value\ Count$ takes the tuples of **Comment** and **Photo** as input, and produces as output tuples like $< 100, \mathbf{Comment}, 20 >$, $< 100, \mathbf{Photo}, 50 >$, $< 31, \mathbf{Photo}, 4 >$, ..., where 100, 31, ... are primary key values for **User**, and $deg(100, \mathbf{Comment}) = 20$, $deg(100, \mathbf{Photo}) = 50$,.... $Value\ Gather$ takes these tuples and outputs tuples like $< 100, 20, 50 >$, ... If *Build Distribution* receives 60 tuples of the form $< x, 20, 50 >$, it will output a tuple $< 20, 50, 60 >$.

4.3.3 Generate Degree

The degree of foreign key value v for each table T_i is generated separately based on the degree distribution. Recall from degree distribution we get such tuple:

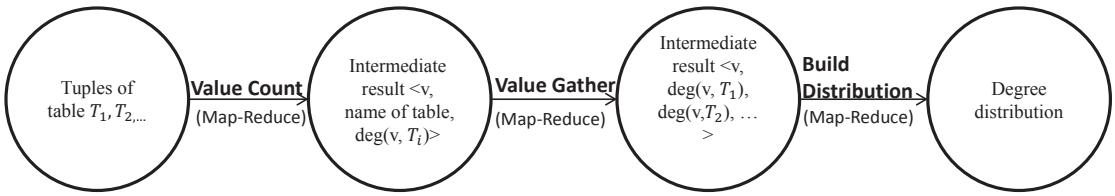

Figure 4.1: Data flow of building degree distribution

$< deg(K, T_1),\ deg(K, T_2), \ldots,\ Fr >$. In degree generation, we generate degree for each particular foreign key value v. For example, we have such a degree distribution tuple: $< deg(K, T_1) = 4,\ deg(K, T_2) = 5,\ Fr = 8 >$, and we know scale factor $s = 10$. When we generate degree for table T_2 on foreign key K, we will generate 80 foreign key values with degree 5. If we have another degree distribution tuple: $< deg(K, T_1) = 7,\ deg(K, T_2) = 5,\ Fr = 6 >$, we will generate another 60 foreign key values with degree 5, when we generate degree for table T_2 on foreign key K.

Since Reducers cannot communicate with each other, it is difficult to generate unique foreign key value in one step before knowing how many tuples each reducer is going to generate. So we do this in two steps. First we generate consecutive foreign key values in each Reducer respectively and record how may values each Reducer has generated in the HDFS file system. We use Reducer id as key value of output, which is used afterwards. Then we add on number of tuples previous Reducers have generated to foreign key values in current Reducer. For example, if $Reducer_0$ and $Reducer_1$ have generated 100 and 200 tuples respectively, $Reducer_2$ will generate tuples with foreign values starting from 301. The tasks can be done in parallel.

Data flow is shown is Fig. 4.3, and pseudo code is shown is Fig. 4.4. For example, we want to generate degree of foreign key **Uid** of **Photo**. The map function of Step 1 takes degree distribution tuples as input, finds $deg(\mathbf{Uid}, \mathbf{Photo})$, and computes new degree frequency using s value as described above. It emits

```
Map (String key, String value)
    //key: tuple id
    //value: tuple

    //Find foreign key value from the tuple.
    //Index of this foreign key is passed via configuration.
    String foreignKeyValue = findForeignKey(value, index);

     //Intermediate result is the foreign key value and "1"
    EmitIntermediate(foreignKeyValue, "1");

Reduce (String key, Iterator values)
        //key: foreign key value
        //values: a list of counts

        int result = 0;
        for each v in values;
                result += ParseInt(v);
        //New key is foreign key value and table name.
        //Table name is passed via. configuration.
        String newKey = key + tableName;

        //Output records foreign key value v,
        //table name of T_i and deg(v, T_i).
        Emit(newKey, result);
```

(a) Value Count

```
Map (String key, String value)
    //key: tuple id
    //value: tuple; <v, deg(v,T_1), deg(v,T_2), …>

    //Find degrees of value v from tuple.
    String degrees = findDegrees(value);
    //Intermediate result is degrees and "1".
    EmitIntermediate(degrees, "1");

Reduce (String key, Iterator values)
        //key: foreign key value: < deg(v,T_1), deg(v,T_2), …,>
        //values: a list of "1"

        int result = 0;
        //Compute the frequency of this degree.
        for each v in values;
                result += ParseInt(v);
        //Output is degree distribution
        // < deg(v,T_1), deg(v,T_2), …, Fr>
        Emit(key, result);
```

(c) Build Distribution

```
Map (String key, String value)
    //key: tuple id
    //value: tuple: <v, T_i, deg(v, T_i)>

    //Find foreign key value from tuple
    String foreignKeyValue = findForeignKey(value);
    //Find table name from tuple.
    String tableName = findTableName(value);
    //Find foreign key degree from tuple.
    String degree = findDegree(value);
    //New value is table name + degree.
    String newValue = tableName + degree

    //Intermediate result records the foreign key value,
    //name of the table having this foreign key
    //and the degree of this foreign key value.
    EmitIntermediate(foreignKeyValue, newValue);

Reduce (String key, Iterator values)
        //key: foreign key value
        //values: a list of table name and degree

        //"degree" records deg(v, T_i) for each table
        //that references K.
        //"numReferencedTable" is passed via. configuration
        int degree [numReferencedTable];
        //Initialize degree[i] to be 0.
        for each degree[i]
                degree[i] = 0;
        for each v in values;
                String tableName = findTableName(v);
                int currentDegree = findDegree(v);
                int index = findIndex(tableName);
                degree[index] = currentDegree;
        //Output uses foreign key value v as key
        //Degree of v as value
        Emit(key, AsString(degree));
```

(b) Value Gather

Figure 4.2: Pseudo code for building degree distribution

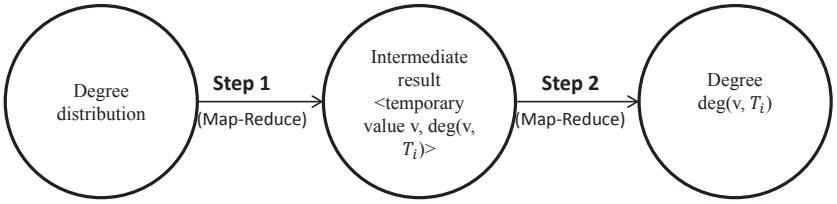

Figure 4.3: Data flow of degree generation

$deg(\mathbf{Uid}, \mathbf{Photo})$ and new degree frequency as intermediate value, and randomly assigns a Reducer to which the intermediate result will be sent. Suppose $Reducer_2$ receives a tuple $< 20, 100 >$, in which 20 is $deg(\mathbf{Uid}, \mathbf{Photo})$ and 100 is frequency of this degree, and has already generated 5000 tuples before it receives this tuple, it will generate such results: $< 5001, 20 >, \ldots, < 5100, 20 >$ and attaches its reducer id with these tuples. After generating all the tuples, the reducer will record how many tuples it has generated in total, and store it into the shared HDFS file system. The map function of Step 2 will take the output of Step 1 as its input and deliver each tuple to the corresponding reducer according to the reducer id attached to each tuple. The reduce function reads in how many tuples each reducer has generated in Step 1 from the file system and computes the add-on value. Suppose $Reducer_0$ and $Reducer_1$ have generated 10000 and 12000 tuples respectively in Step1, the add-on value for $Reducer_2$ will be 22000. Suppose $Reducer_2$ receives a tuple: $< 5001, 20 >$, it will add this add-on value to 5001 and generate the final tuple: $< 27001, 20 >$, which means $degree(27001, \mathbf{Photo})$ is 20.

4.3.4 Compute Dependency Number

Suppose table T depends on table T'. We use tuples from T and T' as input, and compute how many tuples in T have pair $< FK_1,\ FK_2 >$ value that appears in T'. First, we compute number of dependent tuples for each pair, then we sum up

```
Map (String key, String value)
    //key: tuple id
    //value: degree distribution tuple

    //Find degree of current tuple.
    //"index" is passed via configuration.
    int degree = findDegree(value, index);
    //Get frequency of this degree.
    int frequency = getFrequency(value);
    //Set new frequency.
    //"s" is scale factor.
    int newFrequency = frequency * s;
    //Set the Reducer to which the intermediate value is going to be sent
    //"numReducer" is the number of reducers we have
    int reducer = Random.nextInt(numReducer);
    /"/result" value is degree + new frequency
    String result = AsString(degree)+AsString(newFrequency)

    //Intermediate result uses reducer id as key
    //degree and its frequency as value.
    EmitIntermediate(reducer,  result);

Reduce (String key, Iterator values)
        //key: reducer id
        //values: a list of degree and probability

        //"numTuples" records the number of tuples generated
        //in this Reducer.
        long numTuple = 0;
        for each v in values
                int degree = findDegree(v);
                int frequency = findFrequency(v)
                for(int i = 0; i < frequency; i++)
                        //We use numTuples as our temporay foreign key value
                        String result = AsString(numTuples)+ AsString(degree)
                        //Output uses reducer id as its key
                        //temporary foreign key value and its degree as value.
                        Emit(key, result);
                        numTuples++;

    //save number of tuples generated in this Reducer
    writeIntoFile(numTuples)
```

(a) Degree Generation Step 1

```
Map (String key, String value)
    //key: tuple id
    //value: tuples generated from step 1

        //Find the reducer that generate this tuple.
        int reducerID = findReducer(value);
        //Find the temporary foreign key value of this tuple.
        int FKVaue = findFKValue(value);
        //Find degree of this foreign key value
        int degree = findDegree(value);
        String result = AsString(FKValue)+AsString(degree);

        //Intermediate result uses reducer id as key,
        // temporary foreign key value and degree as output value.
        EmitIntermediate(reducerID, result);

Reduce (String key, Iterator values)
        //key: reducer id
        //values: a list of temporary foreign key values and degrees

        //Add on values are passed via configuration
        long[] addons;
        //Get current add on value
        //Current reducer ID is also passed via configuration
        long currentAddon = addons[ReducerID]

        for each v in values
                int degree = findDegree(v);
                //Find temporary foreign key value.
                long FKValue = findFKValue(v)
                //New foreign key value is temporary value plus add on value.
                long newFKValue = FKValue+currentAddon
                //Output is the final foreign key value and its degree.
                Emit(newFKValue, degree);
```

(b) Degree Generation Step 2

Figure 4.4: Pseudo code for degree generation

those number to get the dependency number of table T. The data flow is shown in Fig. 4.5, and the pseudo code is shown in Fig. 4.6.

For example, **Comment** depends on **Photo**. The map function uses the tuples from **Comment** and **Photo** as input. For **Photo**, the primary key **Pid** and foreign key **Uid** values are *pair value*, while for **Comment**, the *pair value*: **Pid** and **Uid** values are both foreign keys. When a tuple arrives into the map function, we determine which file this tuple comes from, if this tuple comes from **Comment**, it is marked as an "A tuple", else it is marked a "B tuple". The *pair value* is retrieved from the tuple and delivered to the reduce function. The reduce function receives "A tuple"s and "B tuple" with the same *pair value*. If a *pair value* exists in a "B tuple", the corresponding "A tuple"s are dependent tuples, else the corresponding "A tuple"s are non-dependent tuples. Note that at most one "B tuple" exists for a particular *pair value*, since the primary key is unique. Suppose the reducer receives a *pair value*: $< 100, 100 >$ and there are 5 "A tuple"s and 1 "B tuple" having this *pair value*, the reducer will store 5 as dependency number for this *pair value*. If the reducer receives a *pair value*: $< 300, 100 >$ and there are 5 "A tuple"s but no "B tuple" having this *pair value*, dependency number for this *pair value* will be 0 and the reduce function will not store it. Dependency number for each single *pair value* is summed up to get the dependency number for **Comment**.

4.3.5 Generate Dependent Degree

Suppose T depends on T' and there are d dependent tuples in T, we need to generate $d * s$ dependent tuples in $\tilde{T}$. First, we get pair $< FK_1,\ FK_2 >$ value $< v_1, v_2 >$ from the generated table $\tilde{T}'$. In order not to break degree distribution of each foreign key, we use $min\{deg(v_1, T), deg(v_2, T)\}$ as *pair degree*. We manage this in three steps with two joins.

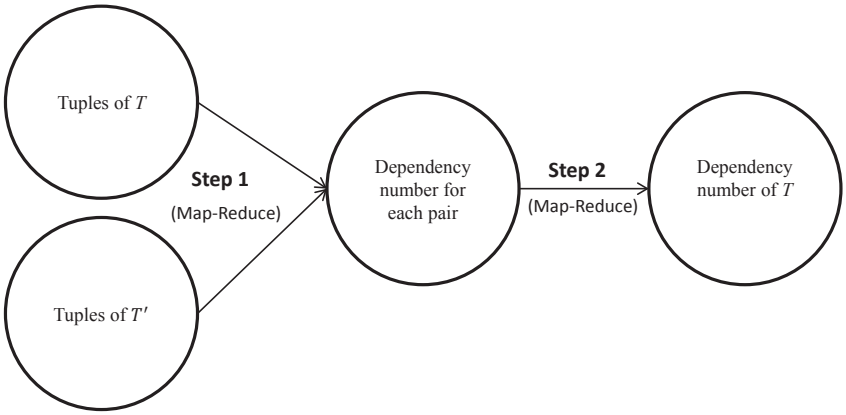

Figure 4.5: Data flow of computing dependency number

```
Map (String key, String value)
    //key: tuple id
    //value: tuple

    //Find which file this tuple is from.
    String tableName = getTableName(value);
    //Get pair value from this tuple.
    Pair pair = getPair(value);
    //Intermediate result is the pair value and which table this pair comes from.
    if(isDependentTable(tableName))
        EmitIntermediate(pair, "A");
    else
        EmitIntermediate(pair, "B");

Reduce (String key, Iterator values)
    //key: pair value
    //values: "A" represents this pair is from T, "B" represents this pair is from T'.

    //"found" records if this pair appears in T'
    boolean found = false;
    //"dependencyNum" records how many times this pair appears in T
    int dependencyNum = 0;
    for each v in values;
        if(v == "A")
            dependencyNum++;
        else if (v == "B")
            found = true;

    //If this pair appears in T' and dependency ratio is not zero, records "dependencyNum"
    if(found && dependencyNum > 0)
        Emit(Null, dependencyNum);
```

Figure 4.6: Pseudo code of computing dependency number

1. Join table $\tilde{T}'$ with tuples $< v_1, deg(v_1, T) >$, in which v_1 is a value of FK_1. In this step, we get the *pair* values and degree of first value in *pair*: $< v_1, v_2, deg(v_1, T) >$.

2. Join the results got from last step with tuples $< v_2, deg(v_2, T) >$, in which v_2 is a value of FK_2. In this step, we get such tuple: $< v_1, v_2, deg(v_1, T), deg(v_2, T) >$.

3. Compute *pair degree* and $left\ degree$ from results we get from last step. We get such tuples in this step: $< v_1, v_2, pair\ degree, left\ degree\ v_1, left\ degree\ v_2 >$.

Since the implementation of step 2 is similar to step 1, we omit its pseudo code. The data flow is shown in Fig. 4.7, and the pseudo code is shown in Fig. 4.8.

For example, **Comment** depends on **Photo**. The map function of Step 1 takes the tuples from $\tilde{\textbf{Photo}}$ and $deg(\textbf{Pid}, \tilde{\textbf{Comment}})$ as input. If a tuple from $\tilde{\textbf{Photo}}$ arrives, **Pid** and **Uid** values are extracted and delivered to the reduce function using **Pid** value as key. If a tuple from $deg(\textbf{Pid}, \tilde{\textbf{Comment}})$ arrives, the **Pid** value and the degree is extracted and delivered to the reduce function using **Pid** value as key. The reducer function does a join operation on **Pid** value, forming a tuple: $<$ **Pid** *value*, **Uid** *value*, $deg(\textbf{Pid}, \tilde{\textbf{Comment}}) >$. Similarly, Step 2 takes tuples generated by Step 1 and $deg(\textbf{Uid}, \tilde{\textbf{Comment}})$, generating tuples: $<$ **Pid** *value*, **Uid** *value*, $deg(\textbf{Pid}, \tilde{\textbf{Comment}}), deg(\textbf{Uid}, \tilde{\textbf{Comment}}) >$. Step 3 takes tuples generated in Step 2 as input, computes *pair degree* as $min\{deg(\textbf{Pid}, \tilde{\textbf{Comment}}), deg(\textbf{Uid}, \tilde{\textbf{Comment}})\}$, $left\ degree$ **Pid** as $deg(\textbf{Pid}, \tilde{\textbf{Comment}}) -$ *pair degree* and $left\ degree$ **Uid** as $deg(\textbf{Uid}, \tilde{\textbf{Comment}}) -$ *pair degree*. Finally, it generates tuples: $<$ **Pid** *value*, **Uid** *value*, *pair degree*, $left\ degree$ **Pid**, $left\ degree$ **Uid** $>$.

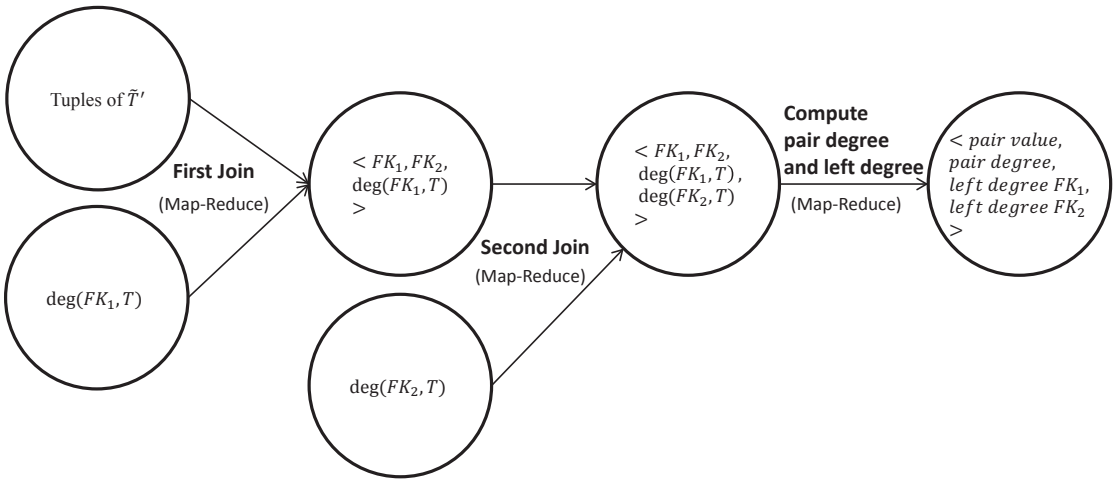

Figure 4.7: Data flow of generate dependent degree

```
Map (String key, String value)
    //key: tuple id
    //value: tuples from T̃′ and deg(FK1, T̃)

    //Find which file this tuple is from.
    String tableName = getTableName(value);
    //Get pair value from this tuple
    Pair pair = getPair(value);
    if(is T̃′(tableName))
        //The PK of T̃′ corresponds to FK1 of T̃.
        String PKValue = findPK(tuple);
        //The FK of T̃′ corresponds to FK2 of T̃.
        String FKValue = findFK(tuple);
        //Use FK1 as key, FK2 as value.
        EmitIntermediate(PKValue, FKValue + "A");
    else
        //Find FK1 of T̃.
        String FKValue = findFK1(tuple);
        String degree = findDegree(tuple);
        //Use FK1 as key, deg(FK1, T̃) as value.
        EmitIntermediate(FKValue, degree + "B");

Reduce (String key, Iterator values)
    //key: FK1 value
    //values: "A value" means the value is FK2 value
    //"B value" means the value is deg(FK1, T̃)

    String FK1Value = key;
    for each v in values;
        if(isAValue(v))
            String FK2Value = getFK2Value(v);
        else if (isBValue(v)")
            String degree = getDegree(v);
    //Output is pair < FK1, FK2 > and deg(FK1, T̃).
    Emit(Null, FK1Value + FK2Value + degree);
```

(a) Dependent Degree Generation Step 1

```
Map (String key, String value)
    //key: tuple id
    //value: tuple

    //Find pair value from tuple.
    String pairValue = findPairValue(value);
    //Find degree of each value of pair from tuple.
    String degrees = findDegree(value);
    //Pair value is key, degrees are value
    EmitIntermediate(pairValue, degree);

Reduce (String key, Iterator values)
    //key: foreign key value
    //values: a list of table name and degree

    //Generate pair degree and left degree.
    for each v in values;
        String pairValue = findPairValue(value);
        int degreeA = findDegreeA(v);
        int degreeB = findDegreeB(v);
        int pairDegree = min(degreeA, degreeB)
        int leftDegreeA = degreeA – pairDegree;
        int leftDegreeB = degreeB – pairDegree;
    //Output uses pair value as key,
    //pair degree and left degree as value.
    Emit(pairValue, pairDegree + leftDegreeA + leftDegreeB);
```

(b) Dependent Degree Generation Step 3

Figure 4.8: Pseudo code for dependent degree generation

```
Map (String key, String value)
//We don't need input in this step.

    //We get numTotalTuples and numReducer from configuration.
    long numTaskTuples = numTotalTuples/numReducer
    for(int i = 0; i < numReducer; i++)
        //Intermediate result uses reducer id as key
        //and number of tuples this reducer needs to generate as value.
        emitIntermediate(i, numTaskTuples);

Reduce (String key, Iterator values)
    //key: reducer id
    //values: number of tuples this reducer needs to generate

    //Compute the starting primary key value.
    long startValue = taskTuples*reducerID;

    for (int i = 0; i < taskTuples; i++)
        String PKValue = AsString(i+startValue);
        String tupleValue = getTuple(PKValue);
        //Output is the tuple content.
        emit(null, tupleValue);
```

Figure 4.9: Pseudo code of generating tables without foreign key

4.3.6 Generate Tables without Foreign Keys

If a table T does not have a primary key, we only need to care about how to generate unique primary key for each tuple. Since we know how many tuples we need to generate, we can tell the number of tuples, which is total number of tuples divided by number reducers, each Reducer needs to generate beforehand. Suppose we need to generate 1000 tuples and we have 10 Reducers, we will assign 100 tuples to each Reducer. $Reducer_0$ generates tuples with primary key value ranging from 0 to 99, $Reducer_1$ 100 to 199, ... We don't need a data flow for this task. The pseudo code is shown is Fig 4.9.

4.3.7 Generate Tables with One Foreign Key

Recall from degree generation step we get such tuples: $< v,\ deg(v, T) >$, in which v is a foreign key value. If we want to generate a table T with only one foreign key, we only need to generate $deg(v, T)$ tuples for each foreign key value v and assign

a unique primary key value for this tuple. Because Reducers cannot communicate with each other, we also need two steps to generate unique primary key value for each tuple, which is similar to degree generation.

1. Generate foreign key value according to degree generated, add a primary key and form a tuple.

2. Adjust primary key value to make it unique.

The data flow is shown in Fig. 4.10 and the pseudo code is shown if Fig. 4.11. For example, **Photo** has one foreign key **Uid**. The map function of Step 1 takes the generated $deg(\mathbf{Uid}, \tilde{\mathbf{Photo}})$ as input, randomly chooses a reducer and sends the **Uid** value and the degree to this reducer. Suppose $Reducer_2$ receives a tuple: $< 200, 20 >$ and has generated 1000 tuples before, it will generate 20 tuples with 200 as their **Uid** value and set the primary key value from 1001 to 1020. The reducer id is also attached with each tuple. After generating all the tuples, each reducer records how many tuples it has generated. In Step 2 the map function deliver the tuples to the corresponding reducer according to the reducer id. The reduce function reads in how many tuples each reducer has generated in Step 1 and computes the add-on value. Suppose $Reducer_0$ and $Reducer_1$ have generated 10000 and 20000 tuples respectively, the add-on value for $Reducer_2$ is 30000. Then $Reducer_2$ will add 30000 to the primary key value of each tuple it receives. Using the content generator, the reduce function get the non-key values and forms a tuple.

4.3.8 Generate Non-dependent Tables with More than One Foreign Keys

We generate non-dependent tables with more than one foreign keys in two steps.

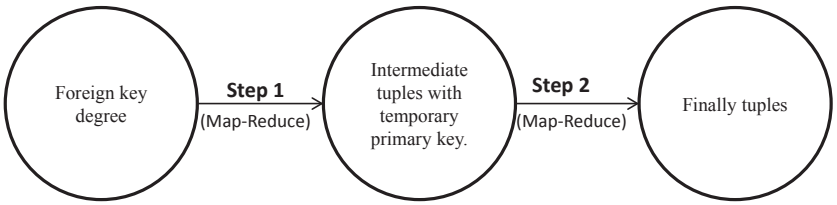

Figure 4.10: Data flow of generating tables with one foreign key

```
Map (String key, String value)
    //key: tuple id
    //value: deg(FK, T̃)

    //Find degree of current tuple.
    int degree = findDegree(value);
    //Get foreign key having this degree.
    String FK = getFK(value);
    //Set the Reducer to which the intermediate value is going to be sent.
    //"numReducer" is the number of reducers we have
    int reducer = Random.nextInt(numReducer);
    //Result value is FK + degree
    String result = AsString(FK)+AsString(degree)
    //Intermediate result uses reducer id as key and result as value.
    EmitIntermediate(reducer,  result);

Reduce (String key, Iterator values)
    //key: reducer id
    //values: a list of degree and foreign key value

    //"numTuples" records the number of tuples generated in this Reducer.
    long numTuples = 0;
    for each v in values
        int degree = findDegree(v);
        String FK = findFK(v)
        for(int i = 0; i < degree; i++)
                //We use numTuples as our primary key value.
                String PK = AsString(numTuples);
                String tupleValue = getTuple(PK, FK);
                //Output is the temporary tuple content.
                Emit(reducerID, tupleValue);
                numTuples++;
    //Save number of tuples generated in this Reducer.
    writeIntoFile(numTuples)
```

(b) Genrate 1 FK Table Step 1

```
Map (String key, String value)
    //key: tuple id
    //value: tuples generated from step 1

    //Find the reducer that generates this tuple.
    int reducerID = findReducer(value);
    //Find the temporary tuple value of this tuple.
    String tupleVaue = findTupleValue(value);

    //We use reducer id as key and  temporary tuple value as output value.
    EmitIntermediate(reducerID, tupleValue);

Reduce (String key, Iterator values)
        //key: reducer id
        //values: a list of temporary foreign key values and degrees

        //Add on values are passed via configuration.
        long[] addons;
        //Get current add on value.
        //Current reducer ID is also passed via configuration.
        long currentAddon = addons[ReducerID];

        for each v in values
                long PKValue = findPKValue(v);
                long FKValue = findFKValue(v);
                long newPKValue = PKValue+currentAddon;
                String newTuple = getTuple(newPKValue, FKValue);
      //Output is the final tuple content.
                Emit(null, newTuple);
```

(a) Generate 1FK Table Step 2

Figure 4.11: Pseudo code for generating tables with one foreign key

1. Generate each foreign key value separately according to the degree we generated, to which we assign a unique primary key value, and then we append the index of this foreign key, which is the index of this foreign key attribute in the tuple. We get such tuples:

$$< primary\ key\ value,\ foreign\ key\ value,\ foreign\ key\ index >$$

   This step is similar to generating tables with one foreign key, so we omit the pseudo code.

2. Join those foreign key values into a tuple. Since the primary key is generated, we can use primary key as key of Map-Reduce task, and set foreign key values according to the foreign key index.

The data flow is shown in Fig. 4.12, and the pseudo code of step 2 is shown is Fig.4.13.

In Flickr we don't have a non-dependent table with more than one foreign key. In TPC-H **PARTSUPP** has two foreign keys **PS_PARTKEYY** and **PS_SUPPKEY**. In Step 1, the generated degree is passed as input, the foreign key value is generated according to the degree and a unique primary key is attached. This is similar to generating one foreign key tables. For example, a degree tuple for **PS_PARTKEY**: $< 100, 20 >$ is received, 10000 tuples have been generated before, and the foreign key index for **PS_PARTKEY** is 1, the output tuple will be: $< 10001, 100, 1 >, \ldots, < 10020, 100, 1 >$. Step 2 does a join operation on the primary key value. Suppose Step 2 receives two tuples: $< 10001, 100, 1 >$ and $< 10001, 7000, 2 >$, it will generate a tuple using 10001 as primary key and set the foreign key to be 100 and 7000 according to the index, forming a tuple $< 10001, 100, 7000, \ldots >$.

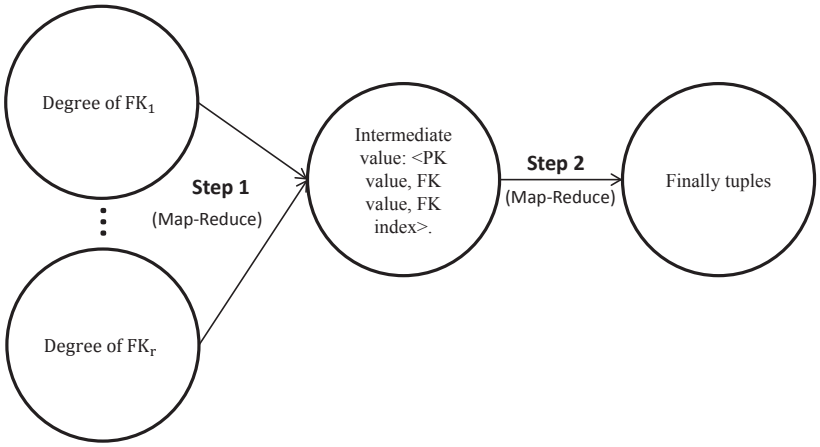

Figure 4.12: Data flow of generating tables with more than one foreign key

```
Map (String key, String value)
    //key is the tuple id
    //value is the tuple generated in last step: <primary key, foreign key, foreign key index>

    //Get primary key as key of the output
    String PKValue = getPKValue(tuple);
    //Get the rest as value of output.
    String left = getLeft(value)
    //Intermediate result uses PKValue as key, FKValue and its index as value.
    emitIntermediate(PKValue, left);

Reduce (String key, Iterator values)
    //key: primary key value
    //values: foreign key value and foreign key index

    //PKValue is key value.
    String PKValue = key;
    String[] FKs;
    for each v in values
        String FKValue = findFKValue(v);
        int index = findIndex(v)
        FKs.add(FKValue, index);
        String tupleValue = getTuple(PK, FKs);
  //Output is tuple conent.
  emit(null, tupleValue);
```

Figure 4.13: Pseudo code of generating tables with more than one foreign key step 2

4.3.9 Generate Dependent Tables with Two Foreign Keys

Since dependent table has dependent tuples and non-dependent tuples, we need to generate them separately, so we manage this in two steps.

1. Generate dependent tuples. Recall we have already generated dependent degree, in which we have foreign key pair and *pair degree*. In this step, we use pair value as foreign keys, and generate tuples according to the *pair degree*.

2. Generate non-dependent tuples. The foreign key values that is not used up in last step are used to generate tuples according to $left\ degree$. We generate each foreign key separately and then merge them together.

Because step 1 is similar to generating tables with one foreign key and step 2 is similar to generating non-dependent table with two foreign keys, we omit the pseudo code. The data flow is shown if Fig. 4.14.

For example, **Comment** depends on **Photo** and has two foreign keys **Pid** and **Uid**, then it is generated in two steps. Step 1 generates the dependent tuples. This step is similar to generating tables with one foreign key, since we can treat the dependent pair as a single foreign key and the *pair degree* as the degree of this foreign key. Suppose 1000000 tuples are generated in Step 1, Step 2 will generate non-dependent tuples with primary key value starting from 1000001. In Step 2, foreign key value **Pid** and **Uid** are generated separately and joined together according to the primary key value attached, which is similar to generating non-dependent tables with more than one foreign key.

4.4 Optimization

Although Map-Reduce can use different nodes to process the input data in parallel, I/O operations are still time consuming. As each task needs to read through the

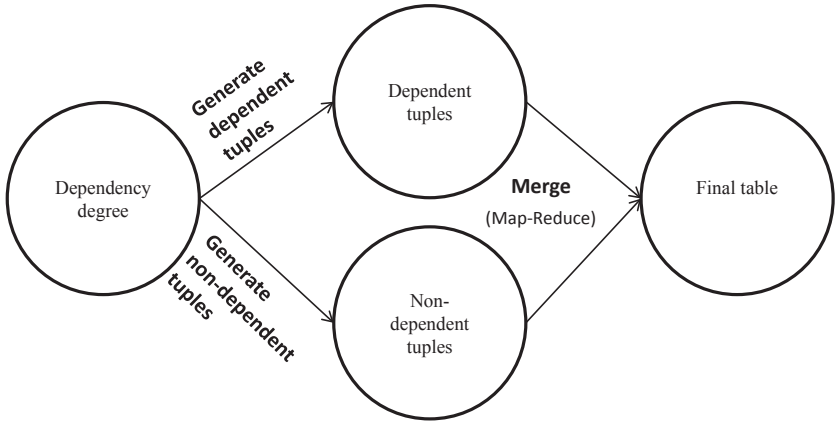

Figure 4.14: Data flow of generating dependent tables with 2 foreign keys

input data once, we must do as much work as we can in one task. Then the number of tasks could be reduced and more time will be saved.

Compute Table Size when Building Degree Distribution

If a table has a foreign key that refers to another table, it must be read once when building degree distribution, during which the table size could be calculated. Each Map-Reduce node stores the number of tuples passed into the map function in a file. After all the nodes finish processing, the number of tuples each node processes will be summed up to get the table size.

Combine Value Count and Value Gather into One Task

Recall in Sec. 4.3.2, we use 3 steps to calculate the degree distribution. In *Value Count* we use one table T_i as input and compute its foreign key degree, getting $< primary\ key\ value\ v,\ name\ of\ T_i,\ deg(v, T_i) >$. In *Value Gather*, using results got from *Value Count* as input, we collect foreign key degrees from each T_i and get $< primary\ key\ value\ v,\ deg(v, T_1),\ deg(v, T_2), \ldots >$. Then we compute degree distribution using the results we got from *Value Gather*. But if we use multiple

tables as input, we can combine *Value Count* and *Value Gather* into one step. We manage this in two phases.

1. In Map phase, we read tuples from each table T_i that refers to T, but each Mapper only reads from one table. In the configuration function, we use a variable called "*foreign key index*" to record the table being read and a variable called "*foreign key sequence*" to find the foreign key value from the tuple. Then the mapper function finds the foreign key value according to *foreign key sequence* and passes it as "key". "Foreign key index" is passed to reducer as "value".

2. In Reduce phase, each Reducer receives tuples having the same foreign key value v. It computes $deg(v, T_i)$ for each table Ti and finally generates $<$ *primary key value* v, $deg(v, T_1)$, $deg(v, T_2), \ldots >$.

The data flow is shown in Fig. 4.15 and the pseudo code is shown in Fig. 4.16. For example, **User** is referred by **Comment** and **Photo**. Suppose the foreign key index of **Comment** is 0 and of **Photo** is 1. The map function of Step 1 will take tuples of **Comment** and **Photo** as input, and produces as intermediate results like $< 100, 0 >$, $< 100, 1 >$, $< 300, 0 >$, ..., in which 100 and 300 are primary key values, 0 and 1 are foreign key indexes. Suppose one Reducer receives 100 as its key, and receives 200 tuples having index 0 and 500 tuples having index 1, it will produce such a tuple as output: $< 100, 200, 500 >$, which means $deg(100, \textbf{Comment})$ is 200 and $deg(100, \textbf{Photo})$ is 500.

Directly Generate Tuples from Degree Distribution

Recall in Sec. 4.3, if we want to generate a table with foreign keys, we must generate degree from degree distribution, and then generate table according to the

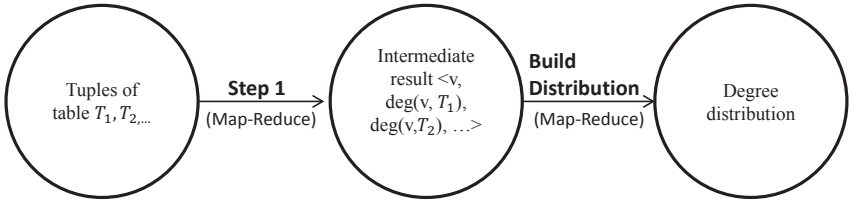

Figure 4.15: Data flow of optimized building degree distribution

```
Map (String key, String value)
    //key: tuple id
    //value: tuples from each table  T_i

    //Find foreign key value from tuple.
    String foreignKeyValue = findForeignKey(value);
    //Set foreign key index as value.
    //Foreign key index is passed via. Configuration.
    String value = index.toString();
    //Intermediate result uses foreign key value as key,
    //foreign key index as value.
    EmitIntermediate(foreignKeyValue, value);

Reduce (String key, Iterator values)
    //key: foreign key value.
    //values: foreign key index.

    //" degree"  records deg(v,  T_i) for each table
    //that references K.
    //" numReferencedTable"  is passed via. configuration
    int degree [numReferencedTable];
    //Initialize degree[i] to be 0
    for each degree[i]
        degree[i] = 0;
    for each v in values;
        int index = Interger.parseInt(value);
        degree[index] += 1;;
    //Output uses foreign key value v as key,
    //degree of v as value
    Emit(key, AsString(degree));
```

Figure 4.16: Pseudo code for optimized building degree distribution step 1

degree generated. Suppose we want to generate a table with n foreign keys, we will need $3 * n + 1$ Map-Reduce tasks: For each foreign key, we need 2 steps to generate degree from degree distribution, as is shown in Fig. 4.3. Besides, we need another $n + 1$ steps to generate tuples from foreign key degrees, as can be seen in Fig. 4.12. However, if we use the following 2 steps to directly generate table from degree distribution, we only need $n + 1$ steps, as is shown in Fig. 4.17.

1. We generate consecutive primary key values and foreign key values in each Reducer according to degree distribution. The output format is <*Reducer id, temporary primary key value and temporary foreign key value*>. We record how many unique primary key values and foreign key values we generated in this Reducer into HDFS file system.

2. In the map function, we compute the add on values for primary key and foreign key according to the Reducer id of the tuple. Then we calculate the final primary key and foreign key value. Intermediate result uses primary key value as key, foreign key value and foreign key index as value. Reducer receives the intermediate result and generate tuple content accordingly.

The data flow is show in Fig. 4.17 and the pseudo code is shown in Fig. 4.18. For example, **PARTSUPP** has two foreign keys: **PS_PARTKEY** and **PS_SUPPKEY**. The map function of Step 1 takes the degree distribution as input. Suppose the scale factor s is 2 and the map function reads in a tuple recording the frequency of $deg(\mathbf{PS\_PARTKEY}, \mathbf{PARTSUPP}) = 5$ is 10. The map function will compute the new frequency is 20 $(2 * 10)$, randomly choose a reducer and send this new frequency and the degree (5) to it. Suppose $Reducer_5$ receives this tuple and it has already generated 1000 primary key (**PARTSUPP** id) values and 200 foreign key values (**PS_PARTKEY**) values before processing it. First, the

reduce function will generate 5 tuples with 201 as foreign key (**PS_PARTKEY**) value and primary key (**PARTSUPP** id) value ranging from 1001 to 1005. Then it will generate another 95 $(10 * 2 * 5 - 5)$ tuples with foreign key value ranging from 202 to 220 and the degree of which is 5. The output tuples are in such format: $< 5, 1001, 201 >, \ldots, < 5, 1005, 201 >, \ldots, < 5, 1100, 220 >$. Suppose the map function receives a tuple $< 5, 1001, 201 >$ and it finds the temporary foreign key value is a **PS_PARTKEY** value according to the file name . First it will find the *PKAddonValue* (number of primary key values previous Reducers has generated) and *FKAddonValue* (number of foreign key values previous Reducers has generated) of $Reducer_5$; suppose they are 100000 and 20000 respectively. Final primary key value and foreign key value are generated accordingly, which are 101001 $(100000 + 1001)$ and 20201$(20000 + 201)$ respectively. Then it will get the foreign key index of **PS_PARTKEY**; suppose it is 1; Intermediate result using primary key value as key, foreign key value and foreign key index as value are passed to the reduce function. The intermediate result is in such format: $< 101001, 20201\&1 >$. Suppose a Reducer receives two tuples with 101001 as key: $< 101001, 20201\&1 >$ and $< 101001, 8888\&2 >$. It will generate a tuple with 20201 as **PS_PARTKEY** value and 8888 as **PS_SUPPKEY** value respectively. The final tuple is in such format: $< 101001, 20201, 8888, \ldots >$.

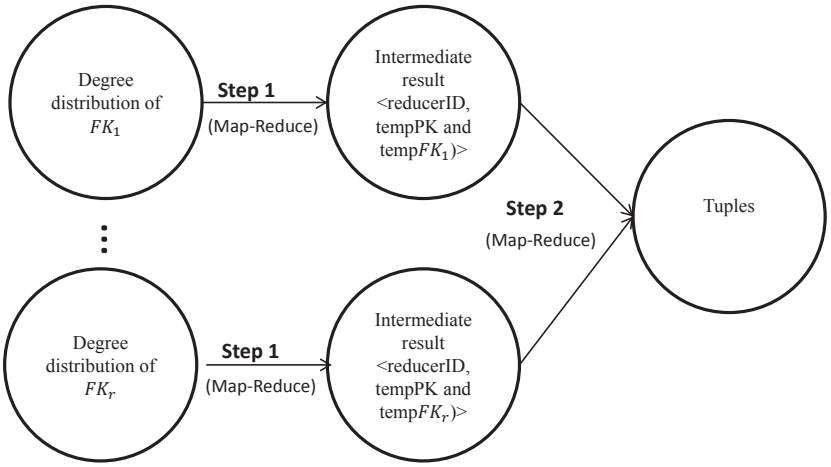

Figure 4.17: Data flow of directly generating non-dependent table from degree distribution

```
Map (String key, String value)
    //key: tuple id
    //value: degree distribution tuple

    //Find degree of current tuple.
    //"index" is passed via configuration.
    int degree = findDegree(value, index);
    //Get frequency of this degree.
    int frequency = getFrequency(value);
    //Set new frequency.
    //"s" is scale factor.
    int newFrequency = frequency * s;
    //Set the Reducer to which the intermediate value is going to be sent
    //"numReducer" is the number of reducers we have
    int reducer = Random.nextInt(numReducer);
    /"/result" value is degree + new frequency
    String result = AsString(degree)+AsString(newFrequency)

    //Intermediate result uses reducer id as key
    //degree and its frequency as value.
    EmitIntermediate(reducer,  result);

Reduce (String key, Iterator values)
        //key: reducer id
        //values: a list of degree and probability

        //"numTuples" records the number of tuples generated
        //in this Reducer.
        long  tempFK= 0;
        long tempPK = 0;
        for each v in values
            int degree = findDegree(v);
            int frequency = findFrequency(v);
            for(int i = 0; i < frequency; i++)
                for(int j = 0; j < degree; j++)
                    String result = AsString(tempPK)+ AsString(tempFK)
                    //Output uses reducer id as its key
                    //temporary primary key and foreign key value as value.
                    Emit(key, result);
                    tempPK++;
                tempFK++;

    //save number of PK and FK generated in this Reducer
    writeIntoFile(tempPK);
    writeIntoFile(tempFK);
```

(a) Generate Non-Dependent Table Step 1

```
Map (String key, String value)
    //key: tuple id
    //value: tuples from Step 1

    Long PKValue = getPKValue(value);
    Long FKValue = getFKValue(value);
    int reducerID = findReducerID(Value);
    //PKAddon and FKAddon is computed according to the reducer id,
    // and is passed via. configuration.
    PKAddon = findPKAddon(reducerID);
    FKAddon = findFKAddon(reducerID);
    Long newPKValue = PKValue + PKAddon;
    Long newFKValue = FKValue + FKAddon;

    //"sequenceNum" records the index of this foreign key attribute,
    //and is passed via. configuration.
    String result = AsString(newFKValue) + AsString(sequenceNum).

    //Intermediate result uses correct PK value as its key,
    //and correct FK value and its sequence as value.
    EmitIntermediate(AsString(newPKValue), result);


Reduce (String key, Iterator values)
        //key: primary key value
        //values: foreign key values and their sequence number.

        Tuple tuple = new Tuple();
        //Set primary key value.
        tuple.setPKValue(key);
        //Set foreign key values according to sequence number.
        for each v in values;
                String FKValue = getFKValue(v);
                int sequenceNum = getSequenceNum(v);
                tuple.addFKValue(FKValue, sequenceNum);
        //Generate non-key values;
        tuple.generateNonKeyValue();

        //Output is tuple content.
        Emit(Null, AsString(tuple));
```

(b) Generate Non-Dependent Table Step 2

Figure 4.18: Pseudo code for directly generating non-dependent table from degree distribution

CHAPTER 5

EXPERIMENTS

In this chapter, we validate UpSizeR by comparing its results against real datasets for various values of s. However, we have no access to any real commercial data from, say, a bank or retailer. We therefore use crawled data from Flickr for comparison. Besides, we also downsize a 40GB TPC-H dataset and compare the results with the dataset generated by DBGen. The performance of optimized and non-optimized UpSizeR is compared using these two datasets. To test the scalability of UpSizeR, we also validate UpSizeR using very large TPC-H datasets.

5.1 Experiment Environment

We conduct our experiment with 10 nodes on the AWAN cluster of our school. For our cluster, each node consists of a X3430 4(4) @ 2.4GHZ CPU running Centos 5.4 with 8GB memory and 2×500G disks. Since our tasks in hand are not computationally intensive, we set number of reducers per node to be 1. Therefore, there are N reducers running on a N-node cluster.

5.2 Validate UpSizeR with Flickr

5.2.1 Dataset

We download four datasets from Flickr for $\mathcal{F}$. These datasets are then combined to give different sizes.

These downloads are at different times. Since $deg(x, \textbf{Photo})$, $deg(x, \textbf{Comment})$ and $deg(x, \textbf{Tag})$ generally increase over time for any user x, the static degree assumption (A3) does not hold. Although we can extend UpSizeR to model the effect, we impose (A3) in this validation exercise by keeping each pair of datasets disjoint through renaming. In other words, if two downloaded datasets $\mathcal{E}_1$ and $\mathcal{E}_2$ have some common **Uid**s (say), we rename the **Uid**s in one of them so that $\mathcal{E}_1$ and $\mathcal{E}_2$ have no common **Uid**s.

Rather than trying to control the scale factor for the real datasets, we let the sizes of real datasets decide the s value for the UpSizeR. Specifically, since the scaling up starts with $\mathcal{D}_0 = \{\textbf{User}\}$, we obtain s by $s = t_1/t_2$, where t_i is the number of **Uid**s in an $\mathcal{F}$ dataset. The baseline size is given by a fixed dataset $\mathcal{F}_{1.00}$ and, in general, $\mathcal{F}$ datasets are denoted as $\mathcal{F}_s$ according to their s value when compared to $\mathcal{F}_{1.00}$. In our case, we have four different scale factors: 1.00, 2.81, 5.35 and 9.11. For example, $\mathcal{F}_{2.81}$ has a number of **Uid**s that is 2.81 times that in $\mathcal{F}_{1.00}$.

5.2.2 Queries

We use five queries to test our UpSizeR. The queries are designed to test whether we have kept the properties we extracted from the empirical dataset.

F1: Retrieve users who uploaded photos. This query is designed for testing the degree distribution property.

#tuples	User	Photo	Comment	Tag	F1	F2	F3	F4	F5
$\mathcal{F}_{1.00}$	146374	529926	1505267	3343964	945	85137	2654	1	820
UpSizeR($\mathcal{F}_{1.00}$, 1.00)	146372	529926	1505264	3343964	944	20378	3114	1	820
$\mathcal{F}_{2.81}$	410892	1557856	4234147	9198476	2398	219499	9717	3	1752
UpSizer($\mathcal{F}_{1.00}$, 2.81)	411305	1589778	4019755	9335860	2137	45537	7282	2	1864
$\mathcal{F}_{5.35}$	783821	2803603	7709470	16299952	4369	401464	15671	4	4096
UpSizeR($\mathcal{F}_{1.00}$, 5.35)	783090	3179552	7932744	17851334	4966	95450	15821	4	4322
$\mathcal{F}_{9.11}$	1332796	4474956	18136861	27743408	8258	734766	27491	15	6645
UpSizeR($\mathcal{F}_{1.00}$, 9.11)	1333448	5299255	13654742	30441367	8741	214662	28302	10	7602

Table 5.1: Comparing table sizes and query results for real $\mathcal{F}_s$ and synthetic UpSizeR ($\mathcal{F}_{1.00}, s$).

F2: Retrieve photographs that are commented on by their owner. This query involves one join, and is designed for testing the dependency ratio property.

F3: Retrieve users who tagged others' photographs. This query involves one join, and is designed for testing the dependency ratio property.

F4: Retrieve users who uploaded photographs but made no comments. This query involves two joins, and is designed for testing the joint distribution.

F5: Retrieve users who write more comments than upload photographs. This query involves two select operation without joins and one select operation with comparison. This query is designed for testing the conditional distribution.

5.2.3 Results

The validation is a comparison between a real $\mathcal{F}_s$ and a synthetic UpSizeR($\mathcal{F}$,s), as is shown in Table 5.1. Consider the size of the tables: when we scale the dataset with $s = 1$, the size of each synthetic table is quite close to the original table. This is because we exactly follow the degree distribution. For the synthetic datasets with $s > 1$, the resulting table size is a little different from the empirical dataset. This is because the degree distribution is not exactly static, breaking the (A3) assumption.

However, the difference between the synthetic dataset and the empirical dataset is within 10%.

Query **F1** shows good results, this is because we exactly follow the degree distribution. The result of query **F2** is not good, this shows that dependency ratio is not well kept. This is because we randomly generate the degree of foreign keys according to degree distribution. And there are not enough dependent pairs for dependent table. The result of query **F3** is better than **F2**, but still not very good, because **F2** and **F3** test the same property. Query **F4** and **F5** give good result, showing that the joint distribution is well kept.

5.3 Validate UpSizeR with TPC-H

5.3.1 Datasets

TPC-H datasets are generated by DBGen and specified by size. The 1GB, 2GB, 10GB and 40GB DBGen datasets are denoted as $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_{10}$ and $\mathcal{H}_{40}$, respectively. We use UpSizeR to scale down $\mathcal{H}_{40}$ with $s = 0.025$, 0.05 and 0.25. Thus, UpSizeR$(\mathcal{H}_{40}, 0.025)$ is a dataset that is similar in size to $\mathcal{H}_1$, and replicate the data correlation extracted from $\mathcal{H}_{40}$.

5.3.2 Queries

The queries we use to compare DBGen data and UpSizeR output are simplified versions of TPC-H queries as shown in Fig. 5.2. The comparison is in terms of number of tuples retrieved and the aggregates computed. All of those queries test the degree distribution property, some of them involve joins on multiple attributes. Since there are no dependent tables in TPC-H dataset, we cannot test this property.

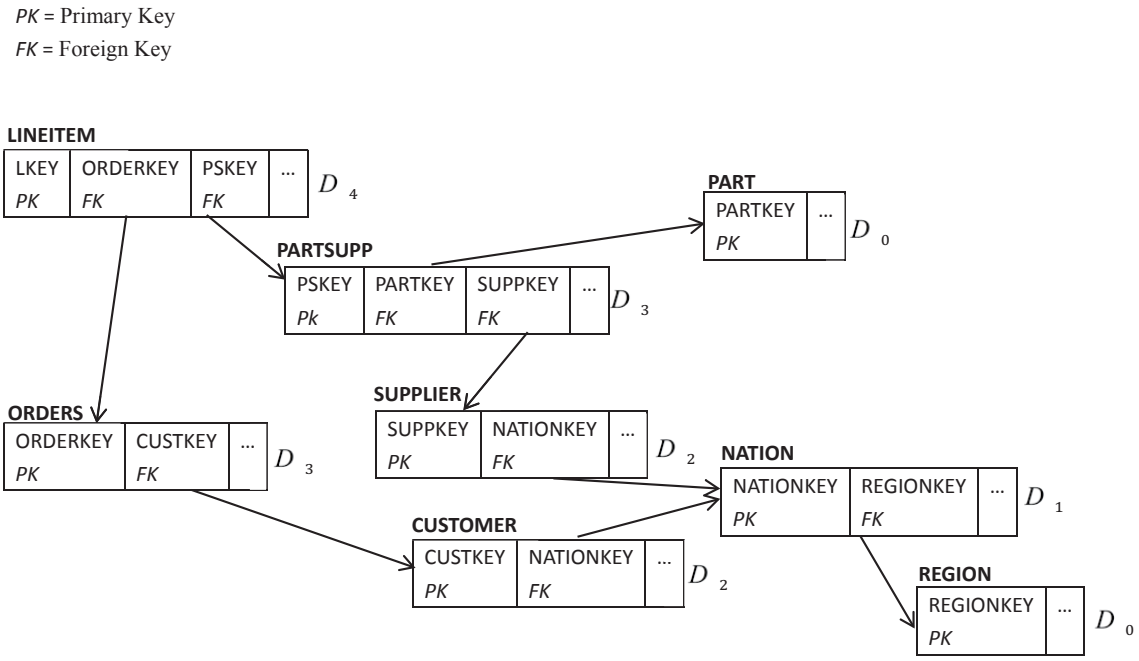

Figure 5.1: Schema $\mathcal{H}$ for the TPC-H benchmark that is used for validating UpSizeR using TPC-H in Sec. 5.3.

5.3.3 Results

Table 5.2 shows good agreement in the number of tuples returned by the queries, which means the degree distribution is well kept when down scaling a dataset. Query H1 computes **ave()** and **count()** and H4 computes **sum()**, so the appropriate comparison is in the returned values. Table 5.3 shows that the aggregates computed with UpSizeR output agrees well with those from DBGen.

5.4 Comparison between Optimized and Non-optimized Implementation

In this section, we compare the time consumed by optimized and non-optimized UpSizeR implementation.

H1:

```
select
    l_returnflag,
    avg(l_extendedprice) as avg_price,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= '1998-12-01'
group by
    l_returnflag
order by
    l_returnflag
```

H2:

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey
from
    part,
    supplier,
    partsupp,
    nation,
    region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and p_size > 21
    and p_type like '%BASS'
order by
    s_acctbal desc,
    n_name,
    s_name,
    p_partkey
```

H3:

```
select
    l_orderkey,
    o_orderdate
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = 'AUTOMOBILE'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
group by
    l_orderkey,
    o_orderdate
order by
    o_orderdate
```

H4:

```
select
    sum(l_extendedprice*(1-l_discount)) as revenue
from
    lineitem,
    partsupp,
    part
where
    (l_ps_id = ps_id
    and ps_partkey = p_partkey
    and p_brand = 'Brand#13'
    and l_shipinstruct like 'DELIVER IN PERSON'
    )
    or
    (l_ps_id = ps_id
    and ps_partkey = p_partkey
    and p_brand = 'Brand#25'
    and l_shipinstruct like 'DELIVER IN PERSON'
    )
    or
    (l_ps_id = ps_id
    and ps_partkey = p_partkey
    and p_brand = 'Brand#35'
    and l_shipinstruct like 'DELIVER IN PERSON'
    )
```

H5:

```
select
    ps_partkey,
    sum(supplycost*ps_availqty) as value
from
    lineitem,
    partsupp,
    supplier
where
    l_ps_id = ps_id
    and ps_suppkey = s_suppkey
    and l_quantity < 20
group by
    ps.ps_partkey
order by
    value desc
```

Figure 5.2: Queries used to compare DBGen data and UpSizeR output

#tuples		H1	H2	H3	H4	H5
1GB	DBGen $\mathcal{H}_1$	3	92196	297453	1	199998
	UpSizeR($\mathcal{H}_{40}$, 0.025)	3	91256	287563	1	199526
2GB	DBGen $\mathcal{H}_2$	3	184156	597099	1	399995
	UpSizeR($\mathcal{H}_{40}$, 0.05)	3	184032	590958	1	399257
10GB	DBGen $\mathcal{H}_{10}$	3	927140	3000540	1	1999983
	UpSizeR($\mathcal{H}_{40}$, 0.25)	3	926152	2995652	1	1999825

Table 5.2: A comparison of resulting number of tuples when query H1,. . . , H5 in Fig. 5.2 are run over TPC-H data generated with DBGen and UpSizeR($\mathcal{H}_{40}$, s), where $s = 0.025, 0.05, 0.25$.

H1 avg(count)						H4
l_returnflag		A	N	R		
1GB	DBGen $\mathcal{H}_1$	38273(1478493)	38248(3043852)	38250(1478870)		6.59E9
	UpSizeR($\mathcal{H}_{40}$,0.025)	38225(1465325)	38265(3043751)	38162(1483526)		6.59E9
2GB	DBGen $\mathcal{H}_2$	38252(2959267)	38234(6076312)	38263(2962417)		1.31E10
	UpSizeR($\mathcal{H}_{40}$,0.05)	38246(2945368)	38287(6075638)	38268(2963548)		1.31E10
10GB	DBGen $\mathcal{H}_{10}$	38237(14804077)	38234(30373792)	38251(14808183)		6.56E10
	UpSizeR($\mathcal{H}_{40}$,0.25)	38268(14803654)	38254(30375214)	38298(14808647)		6.56E10

Table 5.3: A comparison of returned aggregate values: ave() and count() for H1, sum() for H4 shown is Table 5.2 (A, N and R are values of l _returnflag).

5.4.1 Datasets

We scale the dataset we used in Sec. 5.2 and Sec. 5.3. We upsize a 1GB Flickr dataset with scale factor 1.00, 2.81, 5.35 and 9.11 respectively and downsize a 40GB TPC-H dataset with scale factor 0.025, 0.05 and 0.25 respectively.

5.4.2 Results

First we validate the correctness of the dataset we get from optimized UpSizeR. We run the same queries that we use in validating non-optimized UpSizeR on the dataset generated by the optimized UpSizeR, and the results we get are the same as the non-optimized version. This means that optimization does not change the functionality of UpSizeR.

Then we compare the time consumed by both version of UpSizeR. The results

Time	UpSizeR($\mathcal{F}_{1.00}$,1.00)	UpSizeR($\mathcal{F}_{1.00}$,2.81)	UpSizeR($\mathcal{F}_{1.00}$,5.35)	UpSizeR($\mathcal{F}_{1.00}$,9.11)
Non-optimized	14m15s	14m13s	15m15s	15m53s
Optimized	5m20s	5m33s	6m14s	6m52s

Table 5.4: A comparison of time consumed by upsizing Flickr using optimized and non-optimized UpSizeR

Time	UpSizeR($\mathcal{H}_{40}$,0.025)	UpSizeR($\mathcal{H}_{40}$,0.05)	UpSizeR($\mathcal{H}_{40}$,0.25)
Non-optimized	35m13s	35m28s	36m13s
Optimized	18m29s	19m12s	19m25s

Table 5.5: A comparison of time consumed by downsizing TPC-H using optimized and non-optimized UpSizeR

are shown in Table 5.4 and Table 5.5.

From the results we can see that downsizing a big dataset consumes much more time than upsizing a small dataset. This is because if the input dataset is big, more data need to be read from disk and more intermediate result will be generated which will cause more I/O operations. The optimized UpSizeR reduces the read operation. For example, the non-optimized UpSizeR needs to read through the input table files twice: one for computing table size and one for building degree distribution. But optimized version only need to read them once. The optimized UpSizeR also greatly reduces the intermediate results. It can directly generate table contents from degree distribution, omitting the degree generation step. Because of those optimizations, the time consumed decreases by half.

5.5 Downsize and Upsize Large Datasets

One of the reasons why we use Map-Reduce to implement UpSizeR is to make it able to cope with large datasets. So it is necessary to test the scalability of our UpSizeR.

5.5.1 Datasets

Since finding a real empirical dataset large enough is very difficult, we still use TPC-H benchmark to generate datasets for comparison. We totally generate 5 datasets whose size are 1GB, 10GB 50GB, 100GB and 200GB respectively. We upsize the 1GB dataset with scale factor 10, 50, 100 and 200, and compare the resulting datasets with those generated by TPC-H. We also downsize the 200GB dataset with scale factor 0.5, 0.25, 0.05 and 0.005 respectively, and validate the results.

5.5.2 Queries

We use the same queries as the ones used in Sec. 5.3. But because some datasets are to big to be put into normal DBMS and running queries on such datasets are too time consuming, we use Hive[2], a data warehouse system for Hadoop to analyze large datasets, to run the queries. Since Hive has its own SQL-like language HiveQL, we need to translate our queries into HiveQL.

5.5.3 Results

We use optimized version of UpSizeR to reduce intermediate results and save time. First, we upsize $\mathcal{H}_1$ with scale factor $s = 10, 50, 100, 200$. This tests whether UpSizeR can handle large output. Since the input dataset is not big, we don't get a lot of intermediate result tuples and analyzing the input dataset is fast. The comparison of query results run on data generated by DBGen and UpSizeR are shown in Table 5.6 and 5.7. Then we downsize $\mathcal{H}_{200}$ with scale factor $s = 0.005, 0.05, 0.25, 0.5$ to test wether UpSizeR can handle large input. We get a lot of intermediate result tuples and analyzing the input dataset is very slow. The

#tuples		H1	H2	H3	H4	H5
10GB	DBGen $\mathcal{H}_{10}$	3	927140	3000540	1	1999983
	UpSizeR($\mathcal{H}_1$, 10)	3	927562	2996852	1	1999935
50GB	DBGen $\mathcal{H}_{50}$	3	4635650	15002680	1	9999975
	UpSizeR($\mathcal{H}_1$, 50)	3	4634258	14983564	1	9999824
100GB	DBGen $\mathcal{H}_{100}$	3	9270980	30012522	1	19999755
	UpSizeR($\mathcal{H}_1$, 100)	3	9256523	29958632	1	19998373
200GB	DBGen $\mathcal{H}_{200}$	3	18415652	59709948	1	39999525
	UpSizeR($\mathcal{H}_1$, 200)	3	18326525	59625845	1	39985236

Table 5.6: A comparison of resulting number of tuples when query H1,..., H5 in Fig. 5.2 are run over TPC-H data generated with DBGen and UpSizeR($\mathcal{H}_1, s$), where $s = 10, 50, 100, 200$.

H1 avg(count)						H4
l_returnflag		A	N	R		
10GB	DBGen $\mathcal{H}_{10}$	38237(14804077)	38234(30373792)	38251(14808183)		6.56E10
	UpSizeR($\mathcal{H}_1$,10)	38252(14815121)	38265(30364253)	38162(14852321)		6.56E10
50GB	DBGen $\mathcal{H}_{50}$	38252(74020385)	38234(151868960)	38263(74040915)		3.28E11
	UpSizeR($\mathcal{H}_1$,50)	38246(74016423)	38287(151874253)	38268(74125874)		3.28E11
100GB	DBGen $\mathcal{H}_{100}$	38273(147756982)	38248(305733652)	38250(148987700)		6.58E11
	UpSizeR($\mathcal{H}_1$,100)	38544(146963352)	38755(305625440)	38232(148966532)		6.58E11
200GB	DBGen $\mathcal{H}_{200}$	38237(295513964)	38234(611467370)	38251(297975402)		1.32E12
	UpSizeR($\mathcal{H}_1$,200)	38268(294525356)	38254(611525472)	38298(294852563)		1.32E12

Table 5.7: A comparison of returned aggregate values: ave() and count() for H1, sum() for H4 shown in Table 5.6. (A, N and R are values of l _returnflag).

intermediate results are deleted from disk after use to save space. The comparison of query results run on data generated by DBGen and UpSizeR are shown in Table 5.8 and 5.9. From the results we can see that the difference are within 10% for both result size and aggregation value. This means that UpSizeR is able to handle both large input and output.

#tuples		H1	H2	H3	H4	H5
1GB	DBGen $\mathcal{H}_1$	3	92196	297453	1	199998
	UpSizeR($\mathcal{H}_{200}$, 0.005)	3	91322	296544	1	199932
10GB	DBGen $\mathcal{H}_{10}$	3	927140	3000540	1	1999983
	UpSizeR($\mathcal{H}_{200}$, 0.05)	3	928253	2997236	1	1999925
50GB	DBGen $\mathcal{H}_{50}$	3	4635650	15002680	1	9999975
	UpSizeR($\mathcal{H}_{200}$, 0.25)	3	4635235	14983365	1	9999936
100GB	DBGen $\mathcal{H}_{100}$	3	9270980	30012522	1	19999755
	UpSizeR($\mathcal{H}_{200}$, 0.5)	3	9265325	29968535	1	19999963

Table 5.8: A comparison of resulting number of tuples when query H1,..., H5 in Fig. 5.2 are run over TPC-H data generated with DBGen and UpSizeR($\mathcal{H}_{200}, s$), where $s = 0.005, 0.05, 0.25, 0.5$.

H1 avg(count)						H4
l_returnflag		A	N	R		
1GB	DBGen $\mathcal{H}_1$	38273(1478493)	38248(3043852)	38250(1478870)		6.59E9
	UpSizeR($\mathcal{H}_{200}$,0.005)	38268(1480452)	38254(3037325)	38298(1480253)		6.59E9
10GB	DBGen $\mathcal{H}_{10}$	38237(14804077)	38234(30373792)	38251(14808183)		6.56E10
	UpSizeR($\mathcal{H}_{200}$,0.05)	38225(14803655)	38285(30362231)	38136(14802365)		6.56E10
50GB	DBGen $\mathcal{H}_{50}$	38252(74020385)	38234(151868960)	38263(74040915)		3.28E11
	UpSizeR($\mathcal{H}_{200}$,0.25)	38246(74031235)	38287(151573669)	38268(74011977)		3.28E11
100GB	DBGen $\mathcal{H}_{100}$	38273(147756982)	38248(305733652)	38250(148987700)		6.58E11
	UpSizeR($\mathcal{H}_{200}$,0.5)	38235(146534255)	38755(304537517)	38232(148355265)		6.58E11

Table 5.9: A comparison of returned aggregate values: ave() and count() for H1, sum() for H4 shown in Table 5.8 .(A, N and R are values of l _returnflag).

CHAPTER 6

RELATED WORK

As a synthetic dataset generator, UpSizeR's main competitors could be the currently prevalent database benchmarks and other data generation tools. According to our study, most of the dataset generators are vendor-dominated, and could not provide highly customizable data. We study those benchmarks in Sec. 6.1. Due to the domain-specific characteristic, they lack relevance to the real world problems, making them unable to serve their customers well. We thus hear the calling for application-specific benchmarks, which is discussed in Sec. 6.2. We hence see the early signs of application-specific benchmarks as is described in Sec. 6.3. Since we are developing a Map-Reduce version of UpSizeR, we also study a parallel dataset generation tool in Sec. 6.4.

6.1 Domain-specific Benchmarks

The most popular domain-specific benchmarks could be the TPC (Transaction Processing Performance Council) benchmarks. TPC is a non-profit organization founded in 1988 to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry. T-

PC benchmarks are widely used today in evaluating the performance of computer systems. Typically, the TPC produces benchmarks that measure transaction processing (TP) and database (DB) performance in terms of how many transactions a given system and database can perform per unit of time, e.g., transactions per second or transactions per minute. They can provide dataset together with a set of queries to test database management systems.

TPC benchmarks can generate dataset of desired sizes, but each benchmark can only generate dataset in a certain domain, in other words: they are domain-specific: TPC-H for decision support and TPC-W for web transactions, etc. Those benchmarks are designed to provide relevant, objective data, testing method (e.g. queries) and performance metric for academic and industry users to evaluate their products. Vendors can choose a benchmark that fits their applications to compare and improve their products. Researchers can also use them for testing and comparing their algorithms and prototypes. Take TPC-H benchmark as an example, it is a decision support benchmark consisting of a suite of business oriented ad-hoc queries and concurrent data modifications. It represents the industries that need to manage, sell or distribute a product worldwide(e.g., car rental, food distribution, parts, supplier, etc.). The default size of the dataset generated is 1GB and can be scaled up to 100000GB. It also provides a set of queries along with the dataset. The performance metric is query per hour, which measures the number of queries the database system can serve given a specified data size.

Even though the TPC organization was founded more than twenty years ago, it is being adopted by a new generation of benchmarks. Carsten et al. [5] argue that traditional benchmarks (like the TPC benchmark) are not sufficient for analyzing the novel cloud service. They point out five problems of the existing TPC-W benchmark. First, by requiring the ACID properties for data operations, it becomes

obvious that TPC-W has been designed for transactional database system. Second, the primary metric used by TPC-W is the number of web interactions per second (WIPS) that the system under test can handle. Third, the second metric of TPC-W is the ratio of costs and performance ($/WIPS). But the $/WIPS may vary extremely depending on the particular load. Fourth, TPC-W is out of date. Finally, the TPC-W benchmark lacks adequate metrics for measuring the features of cloud systems like scalability, pay-per-use and fault-tolerance. After that, they present some initial ideas on how a new benchmark that fits better to the characteristics of cloud computing should look like. Even though they gave a big picture of a new benchmark that solves the previous problems, the domain-specific feature of TPC benchmarks is still kept.

Similarly, Yahoo! research presents Yahoo! Cloud Serving Benchmark (YCSB) [14] framework, with the goal of facilitating performance comparisons of the new generation of cloud data serving systems. The framework consists of a workload generating client and a package of standard workloads that cover interesting parts of the performance space (read-heavy workloads, write-heavy workloads, scan workloads, etc.). Even though the framework has the extensibility of defining new workload types and the distributions of the operations on the data could be chosen, the dataset of the workload is still domain-specific.

6.2 Calling for Application-specific Benchmarks

Seltzer et al. [30] have already observed the importance of developing application-specific benchmarks considering the irrelevance between the standard domain-specific benchmarks and particular applications. They noted that the result of the existing microbenchmarks or standard macrobenchmarks could provide little

information to indicate how well a particular system can handle a particular application. Such results are, at best, useless and, at worst, misleading. So for database systems, this alternative approach must start with application-specific datasets.

Even if TPC has played a pivotal role in the growth of database industry in the past twenty years, it has a serious short-coming because of the domain-specific nature. Those handful domain-specific benchmarks cannot cover numerous applications, which makes them increasingly irrelevance to the multitude of those data-centric applications. Consider the popular TPC-H benchmark, which can provide a dataset up to 100000GB and a set of queries. Even though it has a rich schema and syntactically complex workloads, it cannot represent tremendous variety of real-world business applications. Moreover, the resulting data is mostly uniform and independent, which makes it impossible to capture the characteristics of the data distribution of the real dataset.

Datasets generated by TPC benchmarks are completely synthetic and domain-specific, since they don't make use of any empirical dataset. Such a method can be traced back to the Wisconsin benchmark [17]. Taking advantage of real data in constructing the benchmark had been considered by its designer. However, they gave up this idea for three reasons: (i) The real dataset must be large enough to reflect its characteristics (such as data distributions, inter- and intra-table relations, etc.). For today's databases, this is certainly not a problem, since some of them are really huge. (ii)If the data is completely synthetic, it is easier to design queries and performance metrics for the benchmarks. So the table sizes and the selectivity could be easily adjusted. This should not prevent us from developing an application-specific benchmark from an empirical dataset either, since the user would already have a set of queries in hand. (iii) There are a lot of difficulties in scaling an empirical dataset. Although 28 years have passed, this third reason remains true.

If we relook at this problem, we will find it is still long overdue. Consider what we should do to scale an empirical dataset. First we need to extract properties from the original data base. This is a very difficult problem, since we must decide what properties to be kept from the original dataset. For each single column of a table, we need to consider the data distribution. Inside each table, we need to consider co-relationship among columns. We also need to take the relationship among tables into consideration. After deciding what properties to retrieve, we still need to consider how to extract and store those properties. The second step, that is injecting those properties into the new database, is more difficult. How to keep all those properties, including the data distribution and inter- and intra- table relationship, is very challenging.

6.3 Towards Application-specific Dataset Generators

So far, the use of empirical datasets is still at a preliminary level in the dataset generation. MUDD (A Multi-Dimensional Data Generator) [31] is a dataset generator designed for TPC-DS, a decision support benchmark being developed by TPC. It is able to generate up to 100 terabyte of flat file data in hours, utilizing modern multi processor architectures, including clusters. It can make use of real data in generating the dataset. However, it extracts only names and addresses, leaving data distribution and column relationships untouched. Similarly, TEXTURE [21] is a micro-benchmark for query workloads, and considers two central text support issues: (i) queries with relevance ranking rather than those that just compute all answers, (ii) a richer mix of text and relational processing, reflecting the trend toward seamless integration. It can extract some properties (such as word distri-

butions, document length etc.) from the "seed" documents. But unfortunately, like how TPC generates tuples, these properties are only *independently* used to generate synthetic documents.

Similarly, Houkjær[24] provides a DBMS independent, and highly extensible relational data generation tool with a graph-model based data-generation algorithm. This seems similar to our UpSizeR. But only cardinalities and value distributions are extracted from the dataset. Since this is only an inter-column property, the correlations among columns and tables are not replicated. In the current industry field, this is also a common practice. Teradata and Microsoft's SQL Server are currently prevalent relational database management systems. They both generates data only use column statistics (such as mode, maximum, distinct vales and number of rows, etc.) While IBM's Optim and HP's Desensitizer [10] don't focus on synthetic data generation, but data extraction and obfuscation.

Bruno and Chaudhuri designed a flexible framework [8] to specify and generate databases that can model data distributions with rich intra- and inter- table correlations. They introduced a simple special purpose language with a functional flavour, called DGL (Data Generation Language). DGL uses the concept of iterators as basic units that can be composed to produce streams of tuples. DGL can also interact with an underlying RDBMS and leverage its well-tuned and scalable algorithms (such as sorting, joins, and aggregates). Hoag and Thompson also present a Parallel General-Purpose Synthetic Data Generator (PSDG) [23]. PSDG is a parallel synthetic data generator designed to generate "industrial sized" datasets quickly using cluster computing. PSDG depends on SDDL, a synthetic data description language that provides flexibility in the types of data generated. In both generator, users need to specify database schema and the data distribution by themselves. This is not suitable for users who do not know the data distribution

of the dataset. Moreover, the correlations between rows and foreign keys can be captured by neither of the languages.

Using queries to guide the data generation can also be seen as an application-specific way of data generation. Binning et al. propose a Reverse Query Processing (RQP) [6]. Reverse query processing (RQP) gets a query and a result as input and returns a possible database instance that could have produced that result for that query. Reverse query processing is carried out in a similar way as traditional query processing. At compile-time, a SQL query is translated into an expression of the relational algebra. This expression is rewritten for optimization and finally translated into a set of executable iterators. At run-time, the iterators are applied to input data and produce outputs. QAGen [7] uses a given query plan with size constraints to generate a corresponding dataset. It takes the query and the set of constraints defined on the query as input, and generates a query-aware test database as output. The generated database guarantees that the test query can get the desired (intermediate) query results as defined in the test case. However, neither of those tools addresses the Dataset Scaling Problem.

Considering the attributes in the real world datasets are commonly correlated, discovering such correlations is very helpful for developing an application-specific dataset generator. CORD [25] (CORrelation Detective via Sampling) is such a tool for automatically discovering "soft" functional dependencies and statistical correlations between columns. It finds the candidates of column pairs that may have useful and interesting dependency relations by enumerating. Meanwhile, it prunes the unpromising candidates using heuristic method. Its primary use is for query optimization, while it could also be used as a data-mining tool. Similarly, CORADD [27] (CORrelation Aware Database Designer) is another tool that can build indexes and materialized views for a set of queries by exploiting correlations

between attributes. Queries are grouped based on the similarity of their predicts and target attributes. They are then used to guide the discovery of correlations between attributes.

As can be seen from CORD and CORADD, more and more attention is paid to find and make use of the correlations among attributes in a dataset. Apart from query optimization, which is the main usage of these two tools, database research on social networks is such a filed that needs the deep understanding of this problem. This is because our interest lies in studying the social interactions (e.g. writing on Facebook walls [36]) among the users, and most of those interactions should be implicitly captured by the correlations of different attribute values, instead of directly finding them from the explicitly declared friend or contact lists. Online social networks should not be overlooked by the designers of an application-specific dataset generation tool, since they are major users of data-centric systems. So it is necessary to have a better understanding of such data.

Tay [32] proposes a tool for application-specific benchmarking. He argues that the TPC's top-down approach of domain-specific benchmark design is obsolete and we should collaborate a bottom-up program to develop dataset generation tools. And then, he presents a solution: scaling an empirical dataset, which is stated as a **Dataset Scaling Problem**. He gives the motivation of scaling an empirical dataset and raises several problems that may be faced. This paper leads to the occurrence of our UpSizeR.

6.4 Parallel Dataset Generation

Synthetic datasets are usually used to evaluate the performance of certain database systems. As sizes of databases are growing to terabytes, or even more, dataset

generation becomes more time consuming than evaluation. In order to speed up data generation and make it more scalable, the generation tool should employ parallelism.

Jim Gray et. al.[22] propose several parallel dataset generation techniques. They first describe how to partition the job into small tasks and fork a process for each task. Then they gives out several solutions for the problems that may be faced in the parallel generation, such as generating dense unique random data, generating indices on random data, generating data having non-uniform distributions, etc. In our UpSizeR implementation, the Hadoop Map-Reduce platform automatically partitions the job and assigns the resulting tasks to each processing node. We also propose our own method of generating dense unique data (such as primary key values etc.), which is described in Chapter 4. Instead of generating certain specified distribution, we capture the data distribution from the empirical datasets and apply them to the synthetic datasets.

Proposed in 1987 and published in 1994, this tool can be seen as an early sign for the parallel dataset generation. The computation model they assumed is a multi-processor computer MIMD (multiple instruction streams and multiple data streams) architecture, in which each processor has a private memory and are connected via a high speed network, the processes can communicate using messages. Such an architecture, however, is still not commonly owned by a small enterprise nowadays. Compared to such expensive multi-processor computers, which always cost millions of dollars, a cluster of normal single-processor computers (such as PCs), which is employed by us, is a more economical choice. Moreover, this tool aims to generate a totally synthetic dataset, like the TPC benchmarks. Even if it provides the algorithms to generate non-dense non-uniform distributions, such as Zipfian and self-similar distributions, it still cannot capture the distributions of the

real data.

CHAPTER 7

FUTURE WORK

7.1 Relax Assumptions

We have 5 assumptions in our UpSizeR's implementation. We can release these assumptions to make the system more practical.

There are some tables with composite primary keys. In the TPC-H example, table **PARTSUPP** has a combination of column **PARTKEY** and **SUPPKEY** as its primary key. This breaks our (A1) assumption. We manage this by creating a new column **PARTSUPP_ID** as the primary key. This works in many scenarios except there are functional dependencies on the primary keys. We have two options to solve this problem. One is re-implementing UpSizeR to make it able to handle composite primary keys. The other is extracting functional dependency as a property of the dataset and applying this property into the synthetic dataset.

We sort the tables into subsets $\mathcal{D}_0$, $\mathcal{D}_1$,... according to (A2). But in real life, there are some datasets with cyclic schema graph. One simple example is self-loop. For example, there is an **Employee** table with employee ID **Eid** as primary key and manager ID **Mid** as foreign key. This defines a management tree. We can extract such a tree from the original dataset and apply it to the synthetic dataset.

Users can also provide their own method to generate such a tree.

According to (A3), we focus on replicating key value correlations. This is because there are already some products that can generate fake non-key attributes, such as TEXTURE[20]. We can employ them in UpSizeR to generate the non-key attributes.

In our UpSizeR's implementation, we only extract the degree distribution and dependency ratio from the empirical dataset, according to (A4). We know that the properties we keep significantly affect the similarity between the empirical dataset and the synthetic dataset. So it is better to extract and keep more dataset properties. We will talk about this in detail in Sec. 7.2.

In (A5), we assume the properties do not change with the dataset size. This assumption may not be true in real life. For example, we assume the degree distribution is static. But users may upload more photos as time goes by. In this case, the degree distribution is not static. To solve this problem, we have to get a degree growth function. Users can provide such a function. We can also derive such a function using data mining technique. According to Wouter et. al.[15]'s study, affects within a human being group become more balanced and clusterable in the course of time. The development of social network can be divided into phases, and the last phase is stable. So we can extract the changing pattern of social network properties in each phase and apply such a pattern into the synthetic dataset.

7.2 Discover More Characteristics from Empirical Dataset

The properties extracted from the empirical dataset significantly affect the similarity between the original dataset and the synthetic dataset generated. The infor-

mation that is not intentionally preserved will surely be lost. As such, we'd better discover as many properties from the original dataset as possible to make the synthetic dataset more similar to the original dataset. If we have the ability to extract enough properties, we can develop a flexible generator, which can be customized by users to choose the properties to be kept when generating the new dataset.

In this thesis, we classified the properties into data distribution, inter-table relationship and intra-table relationship. And in our implementation, we only extracted the degree distribution and dependency ratio. Degree distribution could be considered as a kind of inter-table relationship. Dependency ratio captures both inter- and intra- table relationship. Besides those two properties, we still have a lot of other properties to extract.

One is data distribution: When generating the non-key value attribute, if the attribute is numerical data, we can extract the data distribution of the column. For example, if we have a table that has a column **age**, we can extract the age distribution among the whole population of the table. Then when generating the new dataset, we can follow this distribution.

Another case is co-clustering among columns. Take our *Flickr* dataset as an example: female users are more likely to comment on flowers. This can be reflected on the co-relationship of **Uid** and **Pid** columns in the **Comment** table. Intuitively, we need to co-cluster those two columns. However, we should not do the co-cluster operation according to those two columns but also need to take the related information (e.g. the gender of the corresponding user, etc.)into consideration. Dhillon et. al. provided an Information-Theoretic Co-clustering [18] method. This method clusters each value in the columns into a class by exploiting the clear duality between rows and columns. However, it only makes use of the information of the columns being co-clustered, and is not suitable for our case. Deodhar et.

al. present a parallel simultaneous co-clustering [16] method, which focuses on predictive modelling of multi-relational data such as dyadic data with associated covariates or "side information". In our Flickr example, when we co-cluster the **Pid** and **Uid** columns in the **Comment** table, we can extract user information associated with **Uid** from table **User** and photo information associated with Pid from table **Photo**, then do the co-cluster operation according to these information. We could take advantage of this method in our tool. But we need to first make clear how to figure out those covariates automatically before doing the co-clustering.

However, it is computationally intractable to replicate all the properties we extract. Sometimes, when we want to keep some of the properties we will lose others. So it is a challenging problem that how can we keep as many properties as possible. Another option is to let the user to choose the priority of the properties, so that we can optimize our solution accordingly.

7.3 Use Histograms to Compress Information

In our implementation, when generating the degree distribution, we store the frequency (occurrence) of each degree. This not only takes up a lot of storage, but consumes a lot of time when reading the degree distribution. One of the solution is to use histograms to compress the information. In Ioannidis' paper [26], he gave a brief history of histograms. Histograms were first conceived as a visual aid to statistical approximations. Even today this point is still emphasized in the common conception of histograms: Webster defines a histogram as "a bar graph of a frequency distribution in which the widths of the bars are proportional to the classes into which the variable has been divided and the heights of the bars are proportional to the class frequencies". However, we can use histograms for cap-

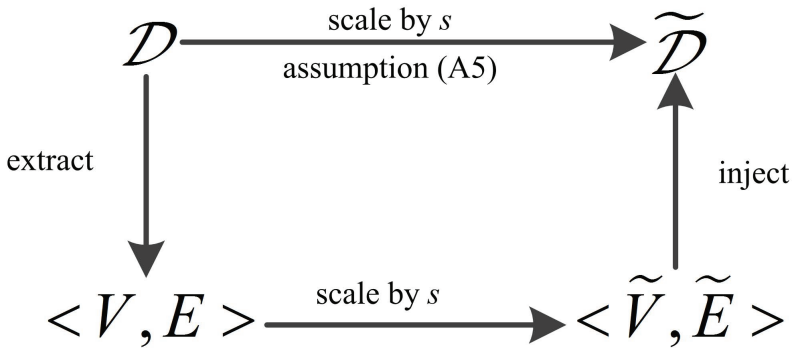

Figure 7.1: How UpSizeR can replicate correlation in a social network database set $\mathcal{D}$ by extracting and scaling the social interaction interaction graph $< V, E >$

turing data distribution approximations even if we don't treat it as a canonical visual representation. For our case, we need to choose a proper multi-dimensional histogram with high efficiency and low information loss. After that, we also need to care about how to eject the compressed information to the synthetic dataset.

7.4 Social Networks' Attribute Correlation Problem

With online social life bringing more and more benefits, researchers are raising stronger and stronger interests in studying the information inherent in social networks. From our observation, upsizing social network data always requires more than classical commercial datasets (in banking, telecom, etc.). In our Flickr example, if two users are friends, they are more likely to comment on each other's photos. Studying such an interaction in $\mathcal{F}$ goes beyond assumption (A5), since it is induced by a social interaction and appears as inter-column and inter-row correlations. How can we design UpSizeR to replicate such correlations?

One possibility is using graph theory [33], as is illustrated in Fig. 7.1: The

social interactions can be represented as a graph $< V, E >$, in which the nodes V represents users in the social network and edges in E represents social interactions among the users. First we extract such a graph from $\mathcal{D}$. Then we scale the graph by s, getting $\tilde{V}, \tilde{E}$. Meanwhile, we generated a synthetic dataset $\tilde{\mathcal{D}}$ under assumption (A5). Finally, we inject $\tilde{V}, \tilde{E}$ into $\tilde{\mathcal{D}}$ by modifying its content.

In extracting the graph $< V, E >$ from a relational database $\mathcal{D}$, the social network interactions to be captured are actually the topology of the graph: such as social triangles (two friends of a friend are likely to be friends), friend path length (6 degrees of separation), etc. In scaling the graph, such a topology must be replicated. In injecting the graph, the social interactions implied in the graph must be reflected as data dependencies in $\tilde{\mathcal{D}}$. A database-theoretic understanding of social networks is required in the graph extraction and injection, while a graph-theoretic understanding is required in the graph scaling. This issue is stated as **Social Networks' Attribute Correlation Problem**[33]:

Suppose a relational database state D records data from a social network. How do the social interactions affect the correlation among attribute values in D's tables?

Many papers are published recently on online social networks. They extract graphs from the social network and study the social interactions. But we found none of them can translate the graph into a relational database. We believe a new and rich area for database research can be explored by this Attribute Correlation Problem.

CHAPTER 8

CONCLUSION

This thesis presents how to implement UpSizeR, a synthetic dataset generation tool, using Map-Reduce. It releases the limitations of previous memory based UpSizeR, making it able to handle the datasets whose size is much bigger than the memory size.

Quite different from normal domain-specific benchmark, UpSizeR generates a dataset of desired size by scaling an empirical dataset. The synthetic dataset generated by UpSizeR keeps the properties of the original one, making it more suitable for testing a database system that will be used for such a dataset. In order to make it able to handle huge datasets and less time consuming, we employ Map-Reduce for our implementation.

The properties extracted from the original dataset determine the similarity between the empirical and synthetic dataset. We discuss what properties to extract and how to use them in Sec. 4.1. Currently we extract and keep three properties: **table size**, **degree distribution** and **dependency ratio**. These properties cover both inter- and intra- table relationship. From this point of view, we propose our UpSizeR algorithm in Sec. 4.2. We use pseudo code and take the Flickr dataset as an example to explain how UpSizeR works in this section. Then we use data

flow and pseudo code to describe the Map-Reduce implementation in Sec. 4.3. For each Map-Reduce task, we use an example to show the input and output format. To make our implementation more efficient, we optimize UpSizeR by combining Map-Reduce tasks. The optimization greatly reduce I/O operations, saving a lot of time.

UpSizeR is validated using Flickr dataset $\mathcal{F}$ and TPC-H dataset $\mathcal{H}$. We upsize $\mathcal{F}$ with scale factor $s > 1$ and downsize $\mathcal{H}$ with $s < 1$. We run certain sets of queries on both the synthetic and the original dataset and compare the results to judge the similarity of both datasets. The results confirm that UpSizeR can approximately scale up table size to be s times the size of the tables in the original dataset. We also compare the time consumed by optimized and non-optimized UpSizeR. The result shows the optimization can greatly reduce time consumption. To test the scalability, we validate UpSizeR using a 200GB TPC-H dataset and also get good results.

Since this is a newly proposed idea, we can rarely find any similar work. However, according to our study, we can hear the calling for application-specific data generator. A lot of researchers find that domain-specific benchmarks cannot meet their needs and the results got from testing on those benchmarks may be useless or even misleading. We hope UpSizeR can open a way for the application-specific benchmarks.

Two major contributions are achieved in this thesis: First, we migrate UpSizeR into Map-Reduce platform to make it more scalable. Second, we optimize its performance to make it more efficient. It is very challenging to attack the Dataset Scaling Problem, considering the explosive increase of the relational databases and the heterogeneity among them. UpSizeR is only a first-cut solution, remaining much to be done. And our implementation employs the now prevalent cloud com-

puting technique to solve the scalability and efficiency problem, making it more innovative than other dataset generation tools. However, this also means a lot of improvements are required to make it a mature product. We have therefore released UpSizeR for open-source development by database community.

BIBLIOGRAPHY

[1] Animoto homepage. `http://animoto.com`.

[2] Hive homepage. `http://hive.apache.org/`.

[3] TPC benchmark homepage. `http://www.tpc.org`.

[4] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack : Facebook's photo storage. *Design*, pages 1–8, 2010.

[5] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather tomorrow? Towards a Benchmark for the Cloud. *Computer*, pages 9:1–9:6, 2009.

[6] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse Query Processing. *2007 IEEE 23rd International Conference on Data Engineering*, pages 506–515, 2007.

[7] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Ozsu. QAGen: Generating Query-Aware Test Databases. *Database*, pages 341–352, 2007.

[8] Nicolas Bruno and Surajit Chaudhuri. Flexible Database Generators. *Order A Journal On The Theory Of Ordered Sets And Its Applications*, pages 1097—1107, 2005.

[9] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):211–222, 2001.

[10] Malu Castellanos, Bin Zhang, Ivo Jimenez, Perla Ruiz, Miguel Durazo, Umeshwar Dayal, and Lily Jow. Data Desensitization of Customer Data for Use in Optimizer Performance Experiments. *Proc Int Conf on Data Engineering ICDE*, pages 1081–1092, 2010.

[11] Ronnie Chaiken, Bob Jenkins, Per-Å ke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE : Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[12] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[13] Surajit Chaudhuri and Vivek R. Narasayya. Automating Statistics Management for Query Optimizers. *IEEE Trans. Knowl. Data Eng.*, 13(1):7–20, 2001.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 143, 2010.

[15] Wouter De Nooy, Andrej Mrvar, and Vladimir Batagelj. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2005.

[16] Meghana Deodhar, Clinton Jones, and Joydeep Ghosh. Parallel Simultaneous Co-clustering and Learning with Map-Reduce. *Granular Computing, IEEE International Conference on*, 0:149–154, 2010.

[17] David J Dewitt. The Wisconsin Benchmark : Past , Present , and Future. pages 1–43, 1981.

[18] Inderjit S Dhillon, Subramanyam Mallela, and Dharmendra S Modha. Information-Theoretic Co-clustering. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining KDD 03*, C(1):89, 2003.

[19] Donko Donjerkovic, Raghu Ramakrishnan, and Yannis Ioannidis. Dynamic Histograms: Capturing Evolving Data Sets. *Data Engineering, International Conference on*, 0:86, 2000.

[20] Vuk Ercegovac, David J DeWitt, and Raghu Ramakrishnan. The texture benchmark: Measuring performance of text queries on a relational dbms. *Proceedings of the 31st international conference on Very large data bases*, pages 313–324, 2005.

[21] Vuk Ercegovac, David J DeWitt, and Raghu Ramakrishnan. The TEXTURE Benchmark: Measuring Performance of Text Queries on a Relational DBMS. *Proceedings of the 31st international conference on Very large data bases*, pages 313–324, 2005.

[22] Jim Gray, Prakash Sundaresan, Susanne Englert, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. pages 243–252, 1994.

[23] Joseph E Hoag and Craig W Thompson. A Parallel General-Purpose Synthetic Data Generator. *ACM SIGMOD Record*, 36(1):19–24, 2007.

[24] Kenneth Houkjæ r, Kristian Torp, and Rico Wind. Simple and Realistic Data Generation. *Database*, pages 1243–1246, 2006.

[25] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658, 2004.

[26] Y. Ioannidis. The History of Histograms (abridged). 2003.

[27] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B Zdonik. CORADD : Correlation Aware Database Designer for Materialized Views and Indexes. *Citeseer*, pages 1103–1113, 2010.

[28] M.E. Nergiz, Chris Clifton, and A.E. Nergiz. MultiRelational k-Anonymity. *IEEE Transactions on Knowledge and Data Engineering*, pages 1104–1117, 2008.

[29] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *The VLDB Journal*, pages 486–495, 1997.

[30] M Seltzer, D Krinsky, and K Smith. The Case for Application-Specific Benchmarking. *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 102–107, 1999.

[31] John M Stephens and Meikel Poess. MUDD: a Multi-Dimensional Data Generator. *ACM SIGSOFT Software Engineering Notes*, 29(1), 2004.

[32] Y. C. Tay. Data Generation for Application-Specific Benchmarking. *PVLDB*, 4(12):1470–1473, 2011.

[33] Y.C. Tay, Bing Tian Dai, Daniel T. Wang, Eldora Y. Sun, Yong Lin, and Yuting Lin. UpSizeR: Synthetically Scaling an Empirical Relational Database. 2010.

[34] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic Multidimensional Histograms. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 428–439, New York, NY, USA, 2002. ACM.

[35] Gary Valentin, Michael Zuliani, and Daniel C. Zilio. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *ICDE*, pages 101–110, 2000.

[36] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. User Interactions in Social Networks and their Implications. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 205–218, New York, NY, USA, 2009. ACM.

[37] Ran Yahalom, Erez Shmueli, and Tomer Zrihen. Constrained Anonymization of Production Data: A Constraint Satisfaction Problem Approach. *Secure Data Management*, pages 41–53, 2011.